Ein Suchwerkzeug für syntaktisch annotierte Textkorpora

Von der Philosophisch-Historischen Fakultät der Universität Stuttgart
zur Erlangung der Würde eines Doktors der
Philosophie (Dr. phil.) genehmigte Abhandlung

Vorgelegt von

Wolfgang Lezius

aus Ibbenbüren

Hauptberichter: Prof. Dr. Christian Rohrer
Mitberichter: apl. Prof. Uwe Reyle

Tag der mündlichen Prüfung: 28. November 2002

Institut für Maschinelle Sprachverarbeitung der Universität Stuttgart

Dezember 2002

Danksagung

Die vorliegende Arbeit wurde unter anderem gefördert durch das DFG-Projekt *Aufbau eines linguistisch interpretierten Korpus des Deutschen* (Projekt TIGER, Projektpartner: Universität des Saarlandes; Universität Potsdam), und das Projekt *Deutsches Referenzkorpus* (Projekt DEREKO, Projektpartner: Universität Tübingen; Institut für Deutsche Sprache, Mannheim), finanziert vom Land Baden-Württemberg.

Diese Arbeit wäre nicht ohne das hervorragende Umfeld des Instituts für Maschinelle Sprachverarbeitung und die Unterstützung zahlreicher Kolleginnen und Kollegen möglich gewesen. An erster Stelle möchte ich mich bei meinem Hauptberichter Christian Rohrer bedanken. Er hatte für die Entwicklung des TIGERSearch-Suchwerkzeugs stets ein offenes Ohr und hat zahlreiche Ideen zur Konzeption des Werkzeugs beigetragen. Ich möchte mich an dieser Stelle insbesondere für seine Ermutigung und Unterstützung bedanken, bereits frühzeitig den Kontakt zur Industrie zu suchen und als Dozent auf industriellen Fachkonferenzen tätig zu werden. Die dort gesammelten Erfahrungen schlagen sich an vielen Stellen in dieser Arbeit nieder.

Ebenfalls bedanken möchte ich mich bei meinem Zweitberichter Uwe Reyle, der viele Ideen zur Überarbeitung dieser Arbeit entwickelt hat. Er hat mir wichtige Anregungen gegeben, insbesondere was die Darstellung der formalen Kapitel der Arbeit betrifft. Besonders hervorzuheben ist sein Engagement und seine Begeisterung, das Suchwerkzeug auch in anderen Kontexten einzusetzen.

Mein besonderer Dank gilt Esther König, die die beiden genannten Projekte und damit speziell auch das TIGERSearch-Teilprojekt betreut hat. Ihr besonderes Engagement um eine theoretische Fundierung des Suchwerkzeugs hat sowohl im Werkzeug selbst als auch in dieser Arbeit zu einem soliden Fundament geführt. Esther stand mir als Ansprechpartnerin immer zur Seite, angefangen von der Konzeption des Suchwerkzeugs über technische Feinheiten im Bereich Java und XML bis hin zu den zahlreichen Korrekturen der Rohfassungen der Dissertation. Besonders bedanken möchte ich mich für den Freiraum, den Esther mir im Rahmen der beiden Projekte stets gelassen hat.

Mein ganz spezieller Dank gilt meinen beiden Kollegen Stefanie Dipper und Arne Fitschen. Sie haben nicht nur unzählige Ideen zur Entwicklung des Suchwerkzeugs beigetragen, sie standen mir in den vergangenen vier Jahren immer mit Rat und Tat zur Seite. Die gemeinsame Arbeit an diversen Projekten, Publikationen und Vorträgen war stets fruchtbar und erfolgreich. Für die Korrekturen der Rohfassungen der Dissertation möchte ich mich ebenfalls ganz herzlich bedanken.

Ein Werkzeug wie TIGERSearch ist immer das Ergebnis einer Teamarbeit. Hier möchte ich vor allem Ciprian Gerstenberger danken, der die Implementation der Benutzeroberfläche mit viel Geduld und Engagement übernommen

hat. Die unzähligen Diskussionen, die wir damit verbracht haben, neue Konzepte für die Oberfläche zu entwickeln bzw. bereits realisierte Konzepte auf den Prüfstand zu stellen, haben sich in vielfacher Hinsicht ausgezahlt. Sehr profitiert hat das Werkzeug von der Diplomarbeit von Holger Voormann zur grafischen Eingabe von Suchanfragen. Vor allem Gelegenheitsbenutzer werden diese Funktionalität zu schätzen wissen. Weiterhin seien die Arbeiten von Hannes Biesinger genannt, der unter anderem dafür gesorgt hat, dass Konvertierungsprogramme für alle gängigen Baumbankformate zur Verfügung stehen.

Damit ein Suchwerkzeug nicht an den Bedürfnissen der Anwender vorbei geht, sind Rückmeldungen von Anwendern und Projektpartnern eine große Hilfe. Hier sind vor allem George Smith von der Universität Potsdam, Tylman Ule von der Universität Tübingen und Detmar Meurers von der Ohio State University zu nennen. Ihre Kommentare haben mich von so manchem Irrweg abgebracht und neue Ideen für die weitere Entwicklung hervorgebracht.

Mein besonderer Dank gilt Manfred Wettler von der Universität Paderborn und Reinhard Rapp von der Universität Mainz, ohne die diese Arbeit nicht entstanden wäre. Sie haben meine früheren Arbeiten an der Universität Paderborn begleitet und mich immer wieder ermutigt, den Weg in die Computerlinguistik zu suchen. Manfred und Reinhard haben mich bei meinen ersten Publikationen hervorragend unterstützt und mit ihrer Begeisterung für die Sache letztlich dafür gesorgt, dass ich mich für die Computerlinguistik entschieden habe.

Abschließend möchte ich mich ganz herzlich bei meiner Familie bedanken, allen voran meiner Frau Ursula, die meine Entscheidung, mich auf das Abenteuer Computerlinguistik einzulassen, von Anfang an mitgetragen hat. Auch meinen Eltern, die mich in diesem Schritt immer bestärkt haben, möchte ich für ihre Unterstützung danken.

Inhaltsverzeichnis

Teil I

Grundlagen

Kapitel 1

Einführung und Motivation

1.1 Vom Textkorpus zur Baumbank

In der Computerlinguistik spielen Korpora, d.h. linguistisch aufbereitete Datensammlungen geschriebener oder gesprochener Sprache, eine zentrale Rolle. Die Verfügbarkeit immer größerer elektronisch lesbarer Korpora hat zu einer eigenen Forschungsrichtung geführt, die als *Korpuslinguistik* bezeichnet wird.

Textkorpora bestehen aus geschriebenen oder transkribierten gesprochenen Texten. Grundeinheiten eines Textkorpus sind die Token, die zu Sätzen zusammengefasst werden[1]. Das erste größere Textkorpus stellte das amerikanische Brown-Korpus dar, das an der Brown University zusammengestellt wurde. Es enthält 500 Texte, die durchschnittlich 2.000 Token umfassen, und wurde 1964 der Forschungsgemeinschaft zur Verfügung gestellt.

Textkorpora mit Wortarten-Annotation

Da ein reines Textkorpus nur begrenzte linguistische Aussagen erlaubt, wurden Textkorpora schon bald annotiert, d.h. mit linguistischer Information angereichert. Die ersten Annotationsexperimente für Textkorpora bestanden in der Zuordnung von Wortartenmarkierungen (so genannter Wortarten-Tags) zu jedem Token. Green und Rubin (1977) annotierten das Brown-Korpus mit Hilfe eines automatischen Verfahrens, dessen Korrektheit 77% betrug. Abb. 1.1 zeigt einen Beispielsatz des Brown-Korpus und seine Wortarten-Annotation.

Das so entstandene Wortarten-annotierte Brown-Korpus stellt bis heute ein häufig verwendetes Textkorpus dar. Es bildete insbesondere die Datengrundlage für die Entwicklung automatischer Verfahren zur kontextsensitiven Wortartenannotation. Ein bekannter Vertreter dieser Arbeiten ist der statistisch ar-

[1] Es sei darauf hingewiesen, dass diese Begriffe Gegenstand linguistischer Diskussion sind. Zur Vereinfachung werden sie in dieser Arbeit aus einer anwendungsbezogenen Sicht verwendet.

Wortform	Wortart
The	AT
Fulton	NP-TL
County	NN-TL
Grand	JJ-TL
Jury	NN-TL
said	VBD
Friday	NR
an	AT
investigation	NN
of	IN
Atlanta's	NP$
recent	JJ
primary	NN
election	NN
produced	VBD
no	AT
evidence	NN
that	CS
any	DTI
irregularities	NNS
took	VBD
place	NN
.	.

Abbildung 1.1: Ein Beispielsatz des Brown Corpus

beitende Wortarten-Tagger Claws, der zur automatischen Annotation des LOB-Korpus (Lancaster-Oslo/Bergen Corpus) entwickelt wurde. Das LOB-Korpus wurde nach denselben Designkriterien wie das Brown-Korpus zusammengestellt und stellt das britische Gegenstück zum Brown-Korpus dar (vgl. Leech et al. 1983). Diese Arbeiten stellen die Vorreiter für die Entwicklung zahlreicher Verfahren zur Wortartenannotation dar, die sowohl statistisch als auch regelbasiert arbeiten (vgl. Church 1988, Cutting et al. 1992, Brill 1995).

Mittlerweile sind vor allem durch die Kombination der Ergebnisse verschiedenartiger Tagger Korrektheitsraten von über 98% erreichbar (vgl. van Halteren et al. 1998). Die Wortarten-Annotation ist damit mit einem geringen manuellen Korrekturaufwand verbunden und stellt heute eine Mindestanforderung an ein Textkorpus dar. Da beim Wortarten-Tagging ein Lexikon bzw. ein Werkzeug zur morphologischen Analyse eingebunden wird, bietet es sich an, zur Wortart auch das Lemma mit anzugeben (vgl. Lezius et al. 1998). Diese automatische Lemmatisierung reichert das Korpus zusätzlich mit einer Lemma-Annotation für jedes Token an. Die Fehlerrate der Lemmatisierung liegt aber mit über 5% deutlich über der Fehlerrate der Wortarten-Annotierung. Eine Einführung in den Themenbereich kontextsensitive Wortartenannotation und automatische Lemmatisierung geben Rapp und Lezius (2001).

Die bekanntesten Textkorpora, die Wortarten- und zum Teil auch Lemma-Annotation umfassen und im Laufe der Zeit Größenordnungen von mehreren 100 Millionen Token angenommen haben, sind meist aus nationalen Initiativen hervorgegangen. Diese Projekte haben den Anspruch, ein repräsentatives Korpus für ihre Sprache zusammenzustellen. Vorreiter für ein solches Referenzkorpus ist das British National Corpus (BNC 2001), das 1994 veröffentlicht

wurde. Vergleichbare Initiativen für amerikanisches Englisch (American National Corpus, ANC 2001) und auch für das Deutsche (Deutsches Referenzkorpus, DEREKO 2001) folgten. Für eine Auflistung verfügbarer Textkorpora sei Barlow (2002) empfohlen. Für eine Diskussion der Kriterien für ein repräsentatives Textkorpus sei auf Evert und Fitschen (2001) verwiesen.

Textkorpora mit syntaktischer Annotation - Baumbanken

Die konsequente Weiterentwicklung von Textkorpora besteht nach dem Wortarten-Tagging in der Annotation der syntaktischen Satzstruktur. Diese Form der Annotation setzt unmittelbar auf dem Wortarten-Tagging eines Korpus auf. Die Anfänge syntaktischer Annotation sind bereits in den 80er Jahren zu finden. Stellvertretend sei hier die Initiative einer Arbeitsgruppe aus Lancaster genannt, die bereits das Wortarten-Tagging für das LOB-Korpus durchgeführt hatte und nun eine syntaktische Annotation des Korpus mit statistischen Methoden durchführte (Garside et al. 1987). Mit der damals zur Verfügung stehenden Hardware konnte jedoch nur ein Bruchteil des LOB-Korpus aufbereitet werden.

Innerhalb weniger Jahre entwickelte sich die syntaktische Annotation umfangreicher Textkorpora zu einem Forschungsschwerpunkt. Entscheidend für diese Entwicklung war die Erkenntnis, dass statistische Parser nur auf der Grundlage großer Datenmengen erfolgreich trainiert und evaluiert werden können (vgl. Marcus et al. 1993). Die in diesem Zusammenhang entstandenen Korpora werden als Baumbanken bezeichnet. Dieser Begriff macht deutlich, dass die Satzstruktur vorzugsweise durch Phrasenstruktur-Bäume ausgedrückt wird. Eine Annotation mit Dependenz-Strukturen ist bis heute nur selten anzutreffen (vgl. aber Prague Dependency Treebank, Hajic 1999). Neben der Entwicklung von Parsern bestehen die Anwendungen von Baumbanken in der Extraktion linguistischer Information (z.B. der Gewinnung von Kollokationen und Subkategorisierungsrahmen) und dem phänomenbasierten Retrieval. Einführungen in den Themenbereich Baumbanken mit den Schwerpunkten englischsprachige und deutschsprachige Baumbanken sind in Leech und Eyes (1997) bzw. Lezius (2001) zu finden.

Die bekannteste Baumbank ist die Penn Treebank (vgl. Marcus et al. 1994 und Taylor et al. 2000). Die Entwicklung dieser Baumbank gliedert sich in drei Phasen. In der ersten Phase (1989-1992) annotierte ein symbolischer Parser die Sätze vor, die anschließend manuell korrigiert wurden. Bedingt durch den hohen manuellen Aufwand musste in dieser Phase ein vereinfachtes Phrasenstruktur-Modell verwendet werden, das auch als Parsing-Skelett bezeichnet worden ist. Die Baumbank umfasste jedoch beachtenswerte 3 Millionen Token und stellte damit eine hervorragende Ausgangsposition für computerlinguistische Anwendungen dar.

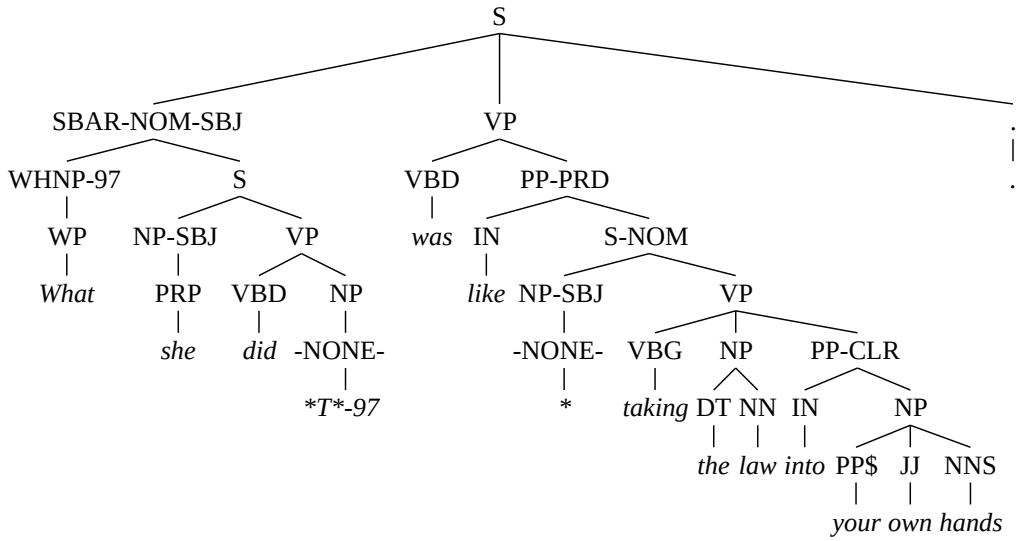


Abbildung 1.2: Ein Beispielsatz der Penn Treebank

Rasch enstand der Wunsch der Anwender der Penn Treebank nach einer detaillierteren Annotation, die u.a. syntaktische Funktionen, Nullkonstituenten und Spuren berücksichtigte (vgl. Marcus et al. 1994). Ein Beispielsatz aus der zweiten Phase der Penn Treebank zeigt, wie diese detaillierte Annotation umgesetzt wurde (vgl. Abb. 1.2). So wird die Bewegung des Frageworts *what* durch eine Spur beschrieben, die die tatsächliche Position (`WHNP`) mit der gedachten Basisposition (`*T*`) über die Index-Nummer `97` koindiziert. Das fehlende Subjekt von *taking the law* wird durch eine Nullkonstituente (`*`) ausgedrückt.

Da eine so detaillierte Annotation nur teilweise durch statistische Methoden unterstützt werden kann, ist der Anteil manueller Korrekturen sehr hoch. Zum Ende der zweiten Phase der Penn Treebank (1993-1995) sind eine Million Wortformen der ersten Phase auf diese Weise aufbereitet worden. In der dritten Phase (1996-1999) ist der Umfang der detaillierten Annotation nochmals um eine Million Token erweitert worden.

Während Abb. 1.2 die Phrasenstruktur des Satzes grafisch darstellt, zeigt Abb. 1.3, wie dieses Datenmodell als so genanntes Klammerstrukturformat auf die Textebene abgebildet wird. Die Textdarstellung eines Korpus wird als Annotations- oder Repräsentationsformat bezeichnet. Eine solche Textdarstellung wird zur Distribution einer Baumbank benötigt. Einen weiteren wichtigen Bestandteil einer Baumbank-Distribution bilden die für die Annotierer aufgestellten Annotationsrichtlinien. Diese auch als Annotationsschema bezeichneten Vorschriften dokumentieren die Eigenschaften der Annotation für die späteren Anwender und stellen zugleich die Konsistenz der Annotation sicher.

```
( (S
    (SBAR-NOM-SBJ
      (WHNP-97 (WP What) )
      (S
        (NP-SBJ (PRP she) )
        (VP (VBD did)
          (NP (-NONE- *T*-97) ))))
    (VP (VBD was)
        (PP-PRD (IN like)
          (S-NOM
            (NP-SBJ (-NONE- *) )
            (VP (VBG taking)
              (NP (DT the) (NN law) )
              (PP-CLR (IN into)
                (NP (PP$ your) (JJ own) (NNS hands) ))))))
    (. .) ))
```

Abbildung 1.3: Klammerstruktur-Darstellung des Beispielsatzes
(What she did was like taking the law into your own hands.)

In den vergangenen Jahren sind Baumbank-Projekte für die meisten europäischen Sprachen initiiert worden. Da eine statistische Annotation noch sehr fehlerbehaftet ist, ist der manuelle Aufwand bei der Annotation sehr hoch. Aus diesem Grunde ist die Größe verfügbarer Baumbanken im Gegensatz zu Korpora, die lediglich auf Wortebene annotiert worden sind (Wortart, Lemma usw.), sehr begrenzt. Eine aktuelle Baumbanken-Übersicht ist in (TIGER Projekt 2002) zu finden. Bekannte deutsche Baumbanken sind die VerbMobil-Baumbank (Hinrichs et al. 2000), in der Dialoge zu Terminvereinbarungen transkribiert und syntaktisch annotiert wurden, sowie die aufeinander aufbauenden Projekte Negra (Skut et al. 1998, Brants et al. 1999) und TIGER (Brants et al. 2002) zur Annotation von deutschen Zeitungstexten.

1.2 Die TIGER-Baumbank

Die vorliegende Arbeit ist im Kontext des Projekts *TIGER* entstanden. Da die Erfordernisse des TIGER-Projekts maßgeblichen Einfluss auf diese Arbeit hatten, werden in diesem Abschnitt die wesentlichen Merkmale der TIGER-Baumbank illustriert. Die Beschreibung lehnt sich an die Darstellung der TIGER-Baumbank in Lezius (2001) an.

Das Projekt TIGER setzt auf den Ergebnissen des Projekts *Negra* auf (Skut et al. 1998, Brants et al. 1999). Im Rahmen des Negra-Projekts ist eine erste deutsche Baumbank annotiert worden, die 20.000 Zeitungssätze umfasst. Die Ziele des TIGER-Projekts sind die Verfeinerung und Erweiterung der Negra-Annotationsrichtlinien (Skut et al. 1997, Brants und Hansen 2002) und die Erstellung einer Baumbank mit 40.000 Sätzen (Brants et al. 2002). Die Annotation erfolgt dabei halbautomatisch. Ein Wortarten-Tagger und ein partieller Parser schlagen für jeden Satz die Wortarten- bzw. Phrasenannotierung vor, die dann vom Annotierer korrigiert werden muss (Brants und Plaehn 2000). Trotz dieser Hilfestellung ist der manuelle Aufwand beträchtlich.

Aus linguistischen Gründen werden im TIGER-Korpus keine reinen Phrasenstrukturen verwendet: Während im Englischen aufgrund der festen Wortstellung Basispositionen (und damit die Position von Spuren) leicht zu bestimmen sind, ist dieses im Deutschen wesentlich schwieriger. Die Basisposition extraponierter Komplementsätze beispielsweise ist bis heute umstritten. Daher sind im TIGER-Korpus Prädikat-Argument-Strukturen als Teile eines ungeordneten Baums (d.h. eines Graphen mit kreuzenden Kanten, vgl. Abb. 1.4) annotiert worden. Für eine ausführliche Begründung dieser Vorgehensweise sei auf Brants et al. (1999) verwiesen.

Die Annotation unterscheidet drei Ebenen (vgl. Abb. 1.4): a) Tags auf Wortebene, die die Wortart und morphosyntaktische Information angeben (z.B. `ART` für Artikel, `Masc.Nom.Sg` für Maskulinum Nominativ Singular); b) Knotenbeschriftungen für phrasale syntaktische Kategorien (z.B. `NP` für Nominalphrase); c) Kantenbeschriftungen für syntaktische Funktionen (z.B. `HD` für Kopf, `SB` für Subjekt).

Kreuzende Kanten werden über die Beschreibung von Argumenten hinaus auch zur Kodierung von weiteren linguistischen Abhängigkeiten verwendet. In Abb. 1.4 liegt beispielsweise ein extraponierter Relativsatz vor, der über eine kreuzende Kante mit der Nominalphrase des Bezugsnomens verknüpft wird.

Eine weitere Besonderheit des Datenmodells stellen die so genannten sekundären Kanten dar. Sie werden u.a. bei der Behandlung der Koordination eingesetzt (vgl. Abb. 1.5). Die koordinierten Elemente werden zusammengefasst, um eine neue Konstituente zu bilden (hier: `CNP` für koordinierte Nominalphrasen und `CS` für koordinierte Teilsätze). Dabei kann es vorkommen, dass Konstituenten in einem der Konjunkte fehlen. In diesem Fall werden sekundäre Kanten gezogen. Im vorliegenden Beispiel ist das Personalpronomen *Er* sowohl Subjekt des ersten als auch des zweiten Konjunkts; *Äpfel und Birnen* ist entsprechend das Akkusativobjekt beider Konjunkte.

Die beschriebene Annotation kann als hybrider Ansatz verstanden werden, der Konstituentenstrukturen mit funktionalen Abhängigkeiten verknüpft, wie sie Dependenzstrukturen ausdrücken. Die in diesem Datenmodell verwendeten Strukturen werden oft als schwache Graphstrukturen bezeichnet, da sie nur in geringem Umfang von der Ausdruckstärke eines Graphen Gebrauch machen,

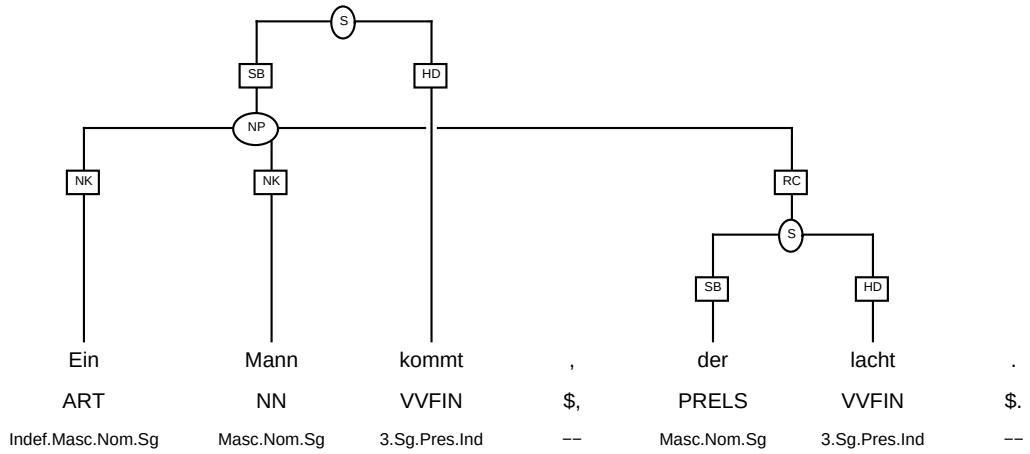


Abbildung 1.4: Annotation von Extraposition durch kreuzende Kanten

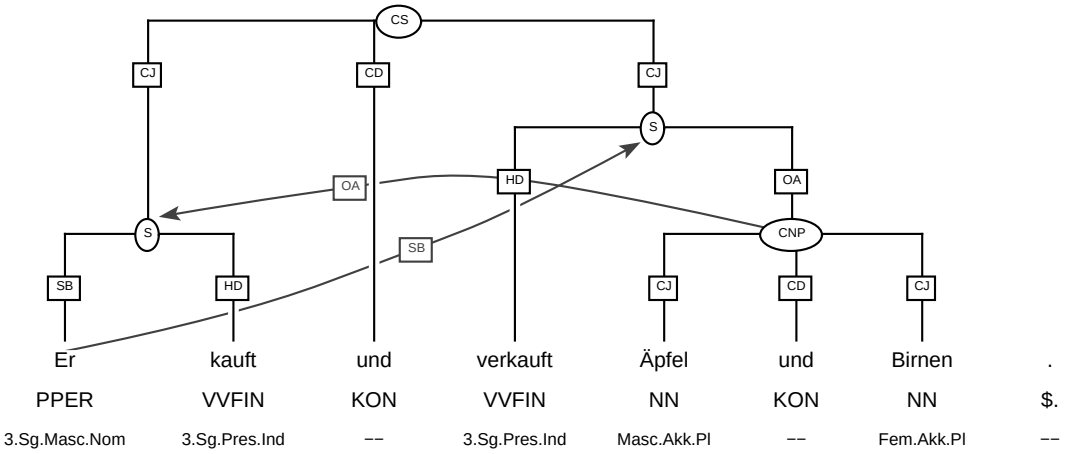


Abbildung 1.5: Annotation von Koordination durch sekundäre Kanten

aber dennoch mächtiger als Baumstrukturen sind. Das verwendete Datenmodell wird in der vorliegenden Arbeit als *Syntaxgraph* bezeichnet. Die in einem Korpus annotierten Syntaxgraphen, die in Kapitel 6 formal definiert werden, haben folgende Eigenschaften:

- Ein Syntaxgraph ist ein gerichteter, azyklischer Graph. [2]
- Jeder Knoten hat einen eindeutig bestimmten Mutterknoten.
- Es gibt genau einen ausgezeichneten Knoten, den Wurzelknoten, der keinen Mutterknoten besitzt.

[2]Die abgebildeten Syntaxgraphen drücken mit Ausnahme sekundärer Kanten die Richtung der Kanten nicht explizit durch Pfeile aus (vgl. Abb. 1.5). Die Richtung der Kanten ergibt sich vielmehr implizit durch die top-down-Anordnung der Knoten.

- Die terminalen Knoten (Wortknoten) sind angeordnet.
- Eine Kante zwischen zwei Knoten kann mit einer Beschriftung versehen sein.
- An jedem Knoten sind Attribut-Wert-Paare angegeben.[3] An allen terminalen bzw. nicht-terminalen Knoten sind stets dieselben Attribute definiert.
- Zusätzlich sind sekundäre Kanten möglich, die zwischen beliebigen Knoten gezogen werden können. Diese Kanten sind gerichtet und dürfen beschriftet sein.

Daneben wird an einen Syntaxgraphen die Forderung gestellt, dass er vollständig disambiguiert ist. Der Wunsch der Korpusannotierer nach einer nicht eindeutigen Annotation ist durchaus verständlich. So könnten in Zweifelsfällen alle denkbaren Alternativen angegeben werden. Doch für die Verarbeitung von Anfragen stellt diese Verallgemeinerung des Datenmodells eine deutliche Verkomplizierung dar. Um die Komplexität der Aufgabe in Grenzen zu halten, werden in dieser Arbeit nur disambiguierte Baumbanken betrachtet.

Es sei an dieser Stelle bemerkt, dass die Einheiten eines Korpus im Laufe dieser Arbeit als Graphen und nicht als Sätze bezeichnet werden. Zwar umfasst ein Korpusgraph in der Regel die Annotation eines Satzes, doch gibt es insbesondere im Bereich der gesprochenen Sprache syntaktisch annotierte Korpora, auf die die Satz-Einteilung nicht zutrifft. Ein Beispiel ist die VerbMobil-Baumbank, die aus so genannten *turns* besteht (vgl. Hinrichs et al. 2000).

Es sei weiterhin bemerkt, dass alle in dieser Arbeit dargestellten Beispiel-Graphen nach dem TIGER-Annotationsschema annotiert worden sind (vgl. Albert et al. 2002).

1.3 Gegenstand und Ziel der Arbeit

Die Menge an Information in einem Textkorpus kann nur mit Hilfe speziell entwickelter Werkzeuge nach linguistischen Phänomenen durchsucht werden. Zur Nutzbarmachung der TIGER-Baumbank wurde daher die Entwicklung eines Suchwerkzeugs benötigt, das die Eigenschaften der Baumbank unterstützt. Ergebnis dieses Teilprojekts ist die vorliegende Arbeit bzw. das im Rahmen dieser Arbeit entwickelte Anfragewerkzeug. Da die entwickelte Software durch die Besonderheiten des TIGER-Projekts maßgeblich beeinflusst ist, wird das Werkzeug *TIGERSearch* genannt.

Da die vorliegende Arbeit im Rahmen des TIGER-Projekts entstanden ist, besteht das vorrangige Ziel in der Entwicklung eines Suchwerkzeugs, das die

[3] Zur besseren Lesbarkeit sind in den abgebildeten Syntaxgraphen keine Attribut-Wert-Paare, sondern lediglich die Attributwerte angegeben.

Eigenschaften des im TIGER-Korpus verwendeten Datenmodells unterstützt. Diese Unterstützung muss sich entsprechend auch in der verwendeten Anfragesprache niederschlagen. Da das Datenmodell auf Graphstrukturen basiert, werden durch diese Forderung zugleich alle mit Hilfe von Baumstrukturen annotierten Baumbanken abgedeckt. Um auch in Zukunft erstellte Baumbanken verarbeiten zu können, sollte das Werkzeug jedoch nicht auf eine feste Menge von annotierten Attributen festgelegt sein, sondern die Verwendung beliebiger Attribute erlauben.

Obwohl die Größe manuell annotierter Baumbanken begrenzt ist, sollte das Werkzeug beliebig große Korpora verarbeiten können. Mit Hilfe statistischer Ansätze können heute Korpora von prinzipiell beliebiger Größe annotiert werden, auch wenn sie nach wie vor mit einer Fehlerrate behaftet sind. Daneben ist die effiziente Verarbeitung von Anfragen eine Grundvoraussetzung für einen effektiven Einsatz eines Suchwerkzeugs. Sie steht aber in dieser Arbeit nicht allein im Mittelpunkt.

Ein weiterer Schwerpunkt dieser Arbeit ist die Entwicklung einer geeigneten Benutzeroberfläche. Sie kann aus einem Anfragewerkzeug ein Explorationswerkzeug machen, das die Visualisierung des Korpus (im Sinne eines Browsers) mit gezielten Suchanfragen kombiniert. Zu den wünschenswerten Hilfsmitteln zählen eine grafische Eingabemöglichkeit für Anfragen sowie eine leistungsfähige grafische Visualisierung der Suchergebnisse.

Zusammengefasst besteht das Ziel dieser Arbeit in der Entwicklung und Implementation eines Suchwerkzeugs, das

- Baumbanken unterstützt, deren Annotation wahlweise auf Baumstrukturen oder auf (schwachen) Graphstrukturen basiert und beliebige Attribute umfassen kann,
- über eine aussdrucksstarke und zugleich leicht erlernbare Anfragesprache verfügt, dessen formale Syntax und Semantik ausführlich dokumentiert sind,
- große Korpora verarbeiten kann,
- effizient arbeitet,
- durch eine intuitive Benutzeroberfläche eine echte Exploration von Baumbanken erlaubt.

Wie Kapitel 3 zeigen wird, ist keines der bislang verfügbaren Werkzeuge in der Lage, all diese Anforderungen zu erfüllen. Das Ziel dieser Arbeit besteht in der Entwicklung eines Werkzeugs, das diese Lücke schließen soll.

1.4 Gliederung der Arbeit

Die Aufgabe, ein Anfragewerkzeug für Baumbanken zu konzipieren und zu implementieren, ist eine Aufgabe des Software-Designs. Der Entwicklungsprozess gliedert sich damit typischerweise in die folgenden Schritte (vgl. Wasserfallmodell in McConnell 1993): Systemspezifikation, Anforderungsanalyse, Architekturdesign, Detaildesign, Implementation, Test und Wartung.

Systemspezifikation und Anforderungsanalyse sind bereits in den beiden vorausgegangenen Abschnitten erfolgt, indem das zu Grunde liegende Datenmodell festgelegt und die Zielanforderungen formuliert worden sind. Während Test und Wartung hier ausgeklammert werden, werden die Schritte Architekturdesign, Detaildesign und Implementation im Laufe dieser Arbeit getrennt für zwei verschiedene Konzeptionsebenen betrachtet: die innere und äußere Konzeption (vgl. Abb. 1.6).

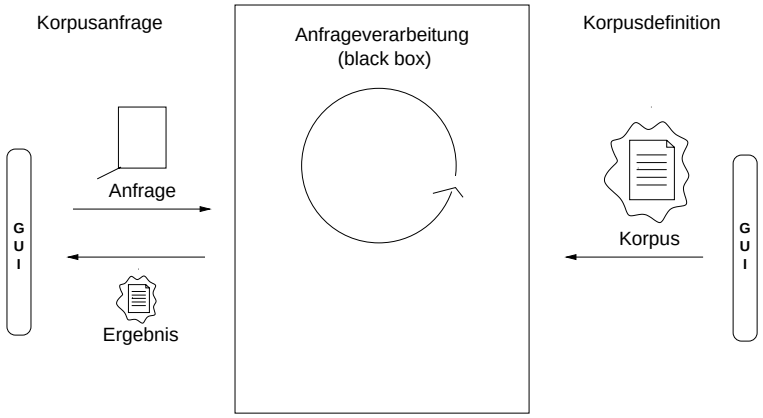


Abbildung 1.6: Innere und äußere Konzeption

Die äußere Konzeption umfasst alle Bereiche, mit denen der Benutzer unmittelbaren Kontakt hat, also die Mensch-Maschine-Schnittstelle. Dazu gehört zum einen das unterstützte Korpuseingangsformat zur Korpusdefinition, denn ohne Definition eines Korpus ist keine Abfrage möglich. Zum anderen zählen hierzu die Anfragesprache, aber auch die Repräsentation der Anfrageergebnisse. Zudem stellt die Gestaltung der Benutzeroberfläche einen zentralen Gesichtspunkt dar.

Die innere Konzeption befasst sich mit Bereichen, die vor dem Benutzer verborgen bleiben, also eine *black box* bilden. Hier findet die Verarbeitung von Anfragen statt. Während die innere Konzeption jederzeit verändert werden kann (sofern das Ein-/Ausgabe-Verhalten der Verarbeitung nicht geändert wird), sollte die äußere Konzeption mit der ersten Veröffentlichung des Werkzeugs möglichst unangetastet bleiben. Als Konsequenz muss die äußere Konzeption mit großer Sorgfalt erfolgen.

Die Gliederung der Arbeit richtet sich nach der Trennung zwischen innerer und äußerer Konzeption:

I. Im ersten Teil dieser Arbeit werden Arbeiten beschrieben, die für die innere und äußere Konzeption relevant sind. Es handelt sich zum einen um Formate zur Kodierung von Baumbanken, zum anderen um bereits verfügbare Suchwerkzeuge für Baumbanken. Diese Arbeiten werden vorgestellt, diskutiert und daraufhin geprüft, welche Ideen im Hinblick auf die hier vorliegende Zielrichtung übertragen werden können.

II. Der zweite Teil der Arbeit konzentriert sich auf die äußere Konzeption. Es geht hier um die Festlegung der Formate für die Korpusdefinition und für die Korpusanfrage. Da der hier verfolgte Ansatz versucht, diese beiden Formate zu einem gemeinsamen zu verbinden, wird dieser Bereich als Korpusbeschreibung bezeichnet.

III. Die innere Konzeption der Anfrageverarbeitung wird im dritten Teil behandelt. Die Anfrageverarbeitung basiert auf einem Verarbeitungskalkül, der zunächst formal definiert und anschließend implementiert wird.

IV. Alle Aspekte der Benutzeroberfläche wie Ansätze zur Visualisierung von Anfrageergebnissen oder ein grafisches Front-End zur Eingabe von Benutzeranfragen werden im vierten Teil dieser Arbeit behandelt.

Den Abschluss der Arbeit bildet die Zusammenfassung und Diskussion.

Kapitel 2

Korpusformate für Baumbanken

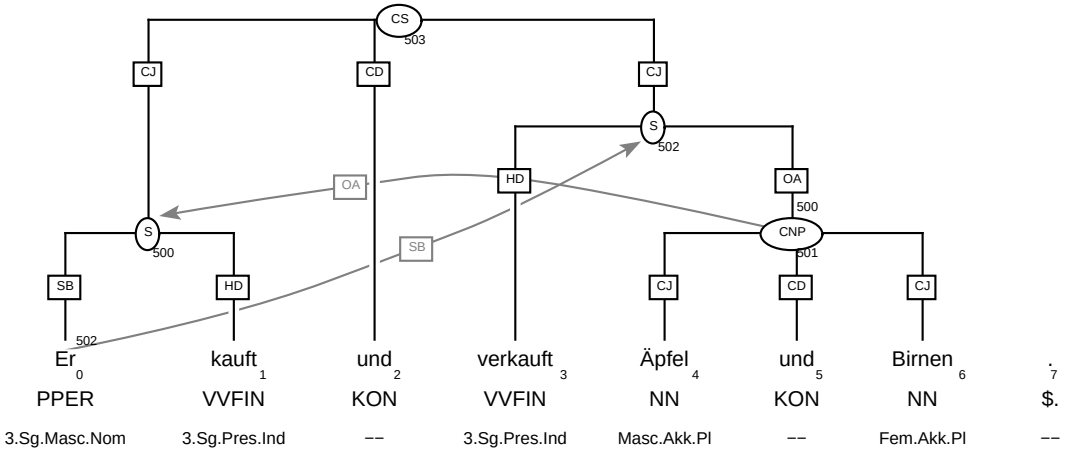
In diesem Kapitel werden Kodierungsformate für Baumbanken vorgestellt, diskutiert und daraufhin untersucht, ob sie sich als Korpusimportformat für das zu entwickelnde Suchwerkzeug eignen. Dabei wird herausgestellt, welche Ideen für diese Arbeit relevant sind.

2.1 Negra-Format

Das Negra-Format ist zur Distribution der deutschsprachigen Negra-Baumbank entwickelt worden (Brants 1997). Da dieses Datenmodell auch im TIGER-Projekt verwendet wird, stellt das Negra-Format eine vollständige Repräsentation des zu unterstützenden Datenmodells dar. Es ist damit auf den ersten Blick ideal als Eingangsformat für das Suchwerkzeug geeignet.

Abb. 2.1 illustriert die Kodierung des Negra-Formats an einem Beispielsatz, der sekundäre Kanten enthält. Dieser Satz wird im Negra-Format durch eine tabellarische Auflistung spezifiziert. Zunächst werden zeilenweise alle Wortknoten, anschließend die übergeordneten Knoten aufgelistet. In jeder Spalte ist ein Attribut kodiert. Die Reihenfolge der Attribute (Wortform, Wortart, morphologische Markierung für äußere Knoten und syntaktische Kategorie für innere Knoten) ist dabei fest vorgegeben. Die angegebene Kantenbeschriftung gilt für die Kante, die vom Elternknoten zum aktuellen Knoten führt. Die Nummer des Elternknoten ist in einer eigenen Spalte angegeben. Durch diese Verweisspalte wird die Graphstruktur repräsentiert. Sekundäre Kanten werden im Anschluss an den Elternknoten aufgelistet. Sie sind durch die Kantenbeschriftung und den Zielknoten eindeutig festgelegt.

Sehr hilfreich ist die Deklaration des Korpus zu Beginn einer Korpusdatei. Alle verwendeten Attribute und Attributwerte werden hier aufgelistet und mit Kurzbeschreibungen erläutert. Eine solche Dokumentation kann von weiterverarbeitenden Anwendungen benutzt werden.



Wortform/ Nummer	Wortart/ Kategorie	morphologische Information	Kanten-label	Eltern-knoten	sek. Kante	Ziel-knoten
Er	PPER	3.Sg.Masc.Nom	SB	500	SB	502
kauft	VVFIN	3.Sg.Pres.Ind	HD	500		
und	KON	-	CD	503		
verkauft	VVFIN	3.Sg.Pres.Ind	HD	502		
Äpfel	NN	Masc.Akk.Pl	CJ	501		
und	KON	-	CD	501		
Birnen	NN	Fem.Akk.Pl	CJ	501		
.	$.	-	-	0		
#500	S	-	CJ	503		
#501	CNP	-	OA	502	OA	500
#502	S	-	CJ	503		
#503	CS	-	-	0		

Abbildung 2.1: Ein Beispielsatz im Negra-Format

Die Vorzüge des Negra-Formats liegen auf der Hand:

- Das Format erlaubt die Kodierung eines Datenmodells, das (schwache) Graphen als Ausdrucksmittel verwendet. Dieses Datenmodell entspricht dem zu unterstützenden Datenmodell der TIGER-Baumbank.

- Daneben erlaubt das Format auch die Kodierung verfügbarer baum-annotierter Textkorpora wie der Penn Treebank.

- Das Format ist sehr kompakt und in Grenzen sogar für den menschlichen Betrachter lesbar. Diese Eigenschaft wird oft als sekundär eingestuft, da die maschinelle Verarbeitung des Formats im Vordergrund steht.

Dabei darf aber nicht vergessen werden, dass es immer noch Anwendungsfälle gibt, in denen der menschliche Benutzer eingreifen muss: Programmierung von Konvertierungsskripten, kurzfristige manuelle Korrekturen an der Annotation usw.

- Durch die Korpusdeklaration sind die Korpuseigenschaften genau definiert und dokumentiert.

Dem gegenüber stehen allerdings auch einige Nachteile, die den Einsatz des Formats erschweren:

- Das Negra-Format ist nicht standardisiert. Die Trennung von Inhalt und Struktur (durch Trennzeichen wie Tabulatoren oder Leerzeichen) ist zwar eindeutig definiert und das Format damit in eindeutiger Weise parsebar. Doch es stehen keine Konvertierungsprogramme für andere Formate wie das Klammerstrukturformat zur Verfügung, so dass jeder Anwender eigene Konverter bzw. Parser programmieren muss.

- Als Zeichensatz wird ISO-Latin1 verwendet. Zeichen aus anderen Zeichensätzen können nicht kodiert werden. Zwar ließe sich hier eine Ersatzkodierung auf der Grundlage von Unicode einbringen (z.B. `\u0937` für das griechische Ω), doch ist auch diese wieder nicht standardisiert.

- Das Format ist nicht erweiterbar. Weitere Attribute wie beispielsweise Lemma auf Tokenebene oder Kasus können nicht ausgedrückt werden.

Für den Einsatz als Eingangsformat des Suchwerkzeugs ist der letzte Punkt entscheidend. Das Negra-Format würde die einsetzbaren Korpora auf einen festgelegten Typ einschränken. Es kommt daher als Eingangsformat nicht in Betracht. Eine wertvolle Anregung ist die Idee der Korpusdeklaration im Dokumentheader.

2.2 Klammerstrukturformate

Das Klammerstrukturformat ist ideal für Baumbanken geeignet, die in Form angeordneter Baumstrukturen annotiert worden sind und pro Knoten nur über ein Attribut verfügen. Die erste Version der Penn Treebank beispielsweise ist nur durch die Attribute Wortart für Token und syntaktische Kategorie für innere Knoten annotiert worden. Zusätzlich zu den annotierten Tags werden eckige oder runde Klammern verwendet, um die Baumstruktur auszudrücken:

```
[S [NP [PN Etta]] [VP [V1 chased] [NP [Det a][N bird]]]]
```

Für einen inneren Knoten werden nach der Angabe der syntaktischen Kategorie (z.B. `S`) alle nachfolgenden Knoten der Reihe nach aufgelistet (für den `S`-Knoten `NP` und `VP`). Die Darstellung der Knoten ist dabei in Klammern eingeschlossen, die die Segmentierung des Satzes ausdrücken. Die Darstellung eines Tokens besteht lediglich aus der Wortklasse und der Wortform (z.B. `N bird`). Die Klammerstrukturdarstellung ist sehr kompakt und beschreibt eine vollständige Baumannotation (vgl. Abb. 2.2). Aus diesem Grunde wird sie in einer Reihe von Baumbanken verwendet (z.B. in einer Variante im Susanne Corpus, vgl. Sampson 1993, 1995).

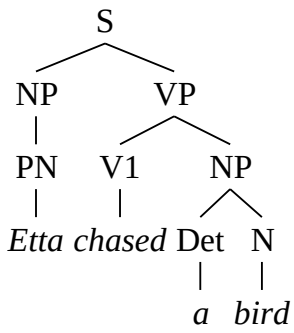


Abbildung 2.2: Ein Beispielsatz im Bracketing-Format

Doch inwieweit lässt sich das in dieser Arbeit verwendete Datenmodell als Klammerstruktur darstellen? Der in Abb. 2.3 abgebildete Beispielsatz, der kreuzende Kanten enthält, soll zur Beantwortung dieser Frage in die Klammerstruktur-Darstellung umgewandelt werden. Satzzeichen bleiben dabei unberücksichtigt.

Ein erstes Problem besteht darin, dass das Bracketing-Format nur genau ein Attribut pro Knoten zulässt. Um diese Einschränkung aufzuweichen, werden die Attributwerte – durch ein ausgezeichnetes Symbol getrennt (in der Regel das Minus-Zeichen) – zu einer Zeichenkette zusammengefasst.

Das viel gravierendere Problem besteht darin, dass das Klammerstrukturformat geordnete Bäume repräsentiert, während im vorliegenden Datenmodell nur die Terminale angeordnet sind und innere Knoten ungeordnet bleiben. Als Folge geht bei der Umwandlung in die Klammerstrukturdarstellung die Anordnung der Terminale verloren und der Relativsatz steht wieder an seiner gedachten Basisposition:

```
(S (NP-SB (ART-NK Ein) (NN-NK Mann)
          (S-RC (PRELS-SB der) (VVFIN-HD lacht))) (VVFIN-HD kommt))
```

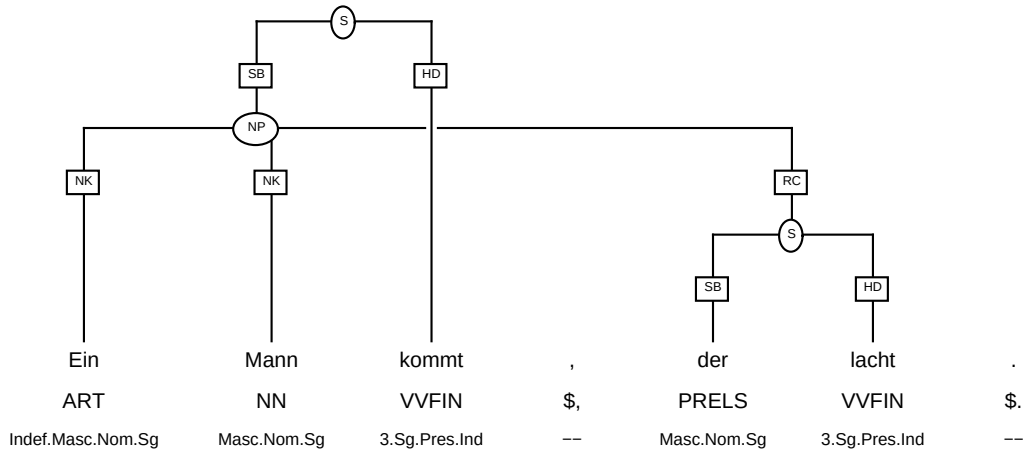


Abbildung 2.3: Ein Beispielsatz mit kreuzenden Kanten

Eine Lösung dieser Problematik besteht darin, die Anordnung der Terminale von außen in die Klammerstruktur zu geben. Dazu wird die Tokenposition als weiterer Attributwert angehängt:

```
(S (NP-SB (ART-NK-1 Ein) (NN-NK-2 Mann)
   (S-RC (PRELS-SB-4 der) (VVFIN-HD-5 lacht))) (VVFIN-HD-3 kommt))
```

Innerhalb der Diskussion um die Problematik kreuzender Kanten wird als eine weitere Lösung immer wieder die Argumentation vertreten, dass kreuzende Kanten zwar zur einfacheren Annotation verwendet werden sollten, die Distribution und Suche jedoch auf einer Variante erfolgen soll, die in Form einer Baumstruktur dargestellt wird (vgl. Ide und Romary 2000a, Abschnitt 3). Um die linguistischen Aussagen der Abhängigkeiten kreuzender Kanten nicht verloren zu geben, wird die erweiterte Klammerstrukturdarstellung nach dem Vorbild der Penn Treebank vorgeschlagen (vgl. Abb. 2.4). Für Korpora, die nach den Annotationsrichtlinien des Negra-Projekts annotiert sind (Brants et al. 1997), hat Thorsten Brants ein Konvertierungsskript entwickelt, das kreuzende Kanten durch Spuren nach den Konventionen der Penn Treebank ausdrückt. Dieses Skript ist Bestandteil der Distribution des Annotations-Werkzeugs *Annotate* (Plaehn und Brants 2000). Abb. 2.4 zeigt, dass die gedachte Basisposition des extraponierten Relativsatzes durch die Spur `*T1*` markiert wird und die hier in Basisposition einzuhängenden Token eine eigene Konstituente `*T1*` bilden.

Der Vorteil einer solchen Lösung besteht darin, dass das Klammerstrukturformat als Repräsentationsmittel beibehalten werden kann. Damit können weiterverarbeitende Programme wie beispielsweise *tgrep* verwendet werden. Doch trotz dieser Vorteile sollte nicht vergessen werden, dass die Entscheidung gegen ein Baummodell und für ein eingeschränktes Graphenmodell mit kreuzenden Kanten eine bewusste linguistische Entscheidung gewesen ist. Wenn eine

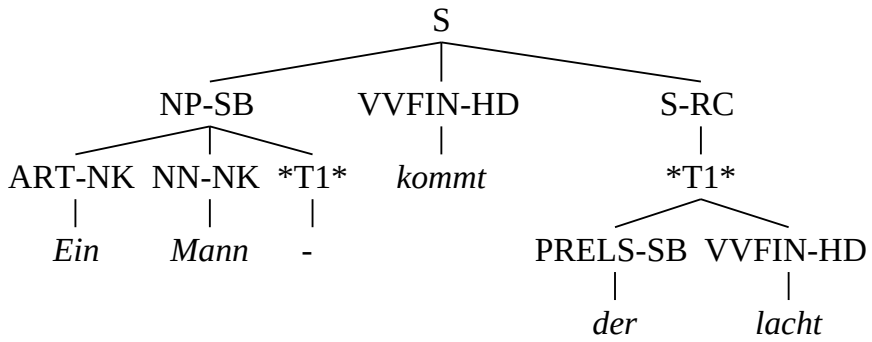


Abbildung 2.4: Ein Beispielsatz mit Spuren

Baumbank sinnvoll nur als Graph annotiert und dargestellt werden kann, so muss auch ein Suchwerkzeug auf diesen Strukturen arbeiten können. Daneben sollte festgehalten werden, dass die Konvertierung der kreuzenden Kanten in eine Spurendarstellung nicht immer so eindeutig ist wie im vorliegenden Fall. Dazu kommt das Problem, dass auch sekundäre Kanten auf ähnliche Weise ausgedrückt werden müssen. Um primäre und sekundäre Kanten unterscheiden zu können, ist eine zusätzliche Markierung notwendig, die die Darstellung noch unübersichtlicher macht. Aus den genannten Gründen ist eine solche Lösung nur begrenzt sinnvoll.

Das Klammerstrukturformat hat insgesamt folgende Vorteile:

- Das Format ist kompakt und platzsparend und bei entsprechender Einrückung sogar für den menschlichen Betrachter gut lesbar.
- Es unterstützt viele gängige Baumbankannotationen.

Die entscheidenden Nachteile sind folgende:

- Zwar ist das Format prinzipiell erweiterbar, indem zusätzliche Attributwerte durch ein Trennsymbol abgetrennt an das Annotationssymbol angehängt werden. Doch ist eine solche Lösung nachteilig, da die einzelnen Attributwerte bei der Weiterverarbeitung wieder isoliert werden müssen. Tritt in einem Attributwert das Trennsymbol auf, muss dieses zusätzlich markiert werden (Trennung von Inhalt und Struktur).
- Als Zeichensatz wird üblicherweise ISO-Latin1 verwendet. Diese Einschränkung kann zwar durch eine Ersatzkodierung aufgehoben werden, doch ist eine solche Lösung nicht standardisiert.
- Das Klammerstrukturformat ist zur Beschreibung von Baumstrukturen gedacht und damit auch auf Baumstrukturen beschränkt. Graphstrukturen

lassen sich nur durch aufwändige Konvertierungen mit zusätzlichen Informationen ausdrücken.

Der Einsatz des Klammerstrukturformats als Eingangsformat für das Suchwerkzeug scheint insgesamt wenig sinnvoll. Unabhängig davon, ob die um Tokenpositionen angereicherte Darstellung oder die Darstellung mit Spuren bevorzugt wird, ist die Darstellung sehr weit entfernt von der ursprünglichen Idee eines ungeordneten Baums. Zudem ist die Erweiterbarkeit des Formats zwar prinzipiell gegeben, aber wenig praktikabel.

2.3 Kodierung syntaktischer Annotation im XCES

Die Initiative des *Corpus Encoding Standard* CES besteht seit 1996. Sie hat sich zur Aufgabe gemacht, einen einheitlichen Kodierungsstandard für linguistisch annotierte Korpora zu entwickeln. Drei Ziele werden mit dieser Initiative verfolgt (vgl. Ide 1998):

- Die Entwicklung von Kodierungsspezifikationen, die die bislang verfolgten Ansätze abdecken, aber zugleich klare Empfehlungen für zukünftige Arbeiten geben.
- Die Minimierung der Kosten für die Erstellung, Annotation und Verwendung von Korpora durch einen einheitlichen Standard.
- Eine höchstmögliche Wiederverwendbarkeit für korpusverarbeitende Werkzeuge, die diesen Standard unterstützen.

Die erste CES-Version basierte vollständig auf der Datenauszeichnungssprache SGML und folgte den Empfehlungen der *Text Encoding Initiative* TEI, die bereits 1994 erste Empfehlungen zur Kodierung linguistischer Korpora gegeben hatte (Barnard und Ide 1997). Mit dem Aufkommen der Datenauszeichnungssprache XML als Nachfolger von SGML wurde der Kodierungsstandard auf XML umgestellt und als XCES weitergeführt (Ide et al. 2000). Im Laufe der Zeit sind die zunächst sehr allgemeinen Spezifikationen immer weiter verfeinert und für spezielle Annotationstypen wie maschinenlesbare Lexika erweitert worden (Ide und Romary 2000b).

XML und XSLT

Die große Unterstützung des Corpus Encoding Standard durch die Forschungsgemeinschaft ist vor allem auf die konsequente Orientierung an aktuellen Technologien zurückzuführen. Insbesondere die Fokussierung auf den Datenauszeichnungsstandard XML zu einem Zeitpunkt, zu dem XML noch kaum verbreitet war, hat sich als Schlüssel zum Erfolg erwiesen. Doch was macht die Datenauszeichnungssprache XML eigentlich aus?

Streng genommen ist XML eine von SGML abgeleitete Sprache, die damit durch benannte Elemente, Elementeinbettung und Elementreferenzen charakterisiert ist (Bray et al. 2000). Im Gegensatz zu SGML ist die Syntax von XML strenger: Öffnende Tags müssen stets geschlossen werden; Attributwerte werden immer in doppelte Anführungsstriche gesetzt; bei Element- und Attributnamen wird Groß-/Kleinschreibung unterschieden. Diese Einschränkungen haben die Entwicklung von Parsern und anderen weiterverarbeitenden Programmen enorm vereinfacht.

Daneben bietet XML komplexere Referenzmechanismen als SGML (XLink, XPointer). XML-Dokumente lassen sich außerdem mit Hilfe von Dokumenttypdefinitionen (DTDs) und insbesondere den XML Schemata (Fallside 2001) sehr detailliert beschreiben und validieren.

Der große Erfolg von XML wird aber auf die Transformationssprache XSLT zurückgeführt, die eine Transformation von XML-Dokumenten in beliebige Zielformate wie HTML erlaubt (Clark 1999). So genannte XSLT-Stylesheets beschreiben, wie das Originaldokument in das Zielformat überführt werden soll. Die durch XSLT-Stylesheets vereinfachte Integration von XML-basierten Ansätzen in das World Wide Web ist ein entscheidender Grund für das große kommerzielle Interesse an XML gewesen, das im Gegenzug zur Entwicklung zahlreicher Verarbeitungswerkzeuge (XML-Editoren, XML-Suchwerkzeuge usw.) geführt hat. Erst eine solche breite Unterstützung macht einen vielversprechenden Ansatz wie XML zu einem Standard. Für eine Einführung in die Technologien XML und XSLT sei auf Fitschen und Lezius (2001a, b) verwiesen.

Für die Kodierung von Textkorpora ist XML als Markup-Sprache eine ideale Grundlage (vgl. Ide 2000a, b): Zeichen außerhalb des gewählten Zeichensatzes lassen sich auf standardisierte Weise durch ihre Unicode-Kodierung ausdrücken, Inhalt und Struktur sind strikt getrennt. Ein XSLT-Stylesheets kann aus einer komplexen Korpuskodierung die gerade fokussierten Daten ausschneiden und auf eine adäquate Weise darstellen. Kodierung und Visualisierung des Korpus sind dadurch getrennt. Daneben erlauben die umfassenden Referenzmechanismen von XML eine sehr flexible Art der Korpusannotation, die als *standoff* Annotation bezeichnet wird. Dieser Ansatz wird im Folgenden vorgestellt.

Stand-off Annotation

Ein immer wiederkehrendes Problem bei der Kodierung von Korpora besteht darin, dass die gewählte Darstellung eine Erweiterung der Annotation nur begrenzt zulässt. So kann beispielsweise eine Baumbank, die im Klammerstrukturformat repräsentiert ist, nicht ohne Änderungen des Repräsentationsformats um eine Lemma-Information erweitert werden.

Eine konsequente Lösung dieser Problematik besteht darin, jede Annotationsebene getrennt zu verwalten. Im Falle einer Baumbank würde dies be-

deuten, in einer primären Datei den Originaltext, in einer weiteren Datei die Wortart- und ggf. Lemma-Annotation, und in einer dritten Datei die syntaktische Annotation zu kodieren. Durch Referenzen auf die primäre Korpustextdatei werden dann die Bezüge zwischen den Annotationsebenen hergestellt (vgl. nächsten Unterabschnitt).

Die erweiterten Referenzmechanismen von XML unterstützen den Ansatz der Verteilung der Daten durch ausdrucksstarke Konzepte (XLink und XPointer, vgl. Ide 2000b). Ein weiterer Vorteil der Trennung von Basis- und Annotationsdaten beim Aufbau eines Korpus besteht darin, dass die Annotierer zeitgleich an verschiedenen Annotationsebenen auf denselben Basisdaten arbeiten können (Carletta et al. 2002). Daneben lässt sich das Korpus später problemlos erweitern. Eine zusätzliche Annotationsebene muss dabei nicht notwendigerweise Bezug zu den Basisdaten nehmen, sondern kann auch an eine bereits bestehende Annotationsebene angehängt werden (z.B. an die Lemma-Annotation).

Syntaxkodierung in XCES

Die Kodierung syntaktischer Annotation wie sie in XCES vorgeschlagen wird (vgl. Ide und Romary 2000a, 2001) basiert auf dem Ansatz der stand-off Annotation. Am Beispiel eines einfachen Satzes, dessen syntaktische Annotation ohne kreuzende Kanten auskommt, soll diese Vorgehensweise zunächst illustriert werden (vgl. Abb. 2.5). Ein weiteres Beispiel wird dann die Probleme dieses Ansatzes aufzeigen.

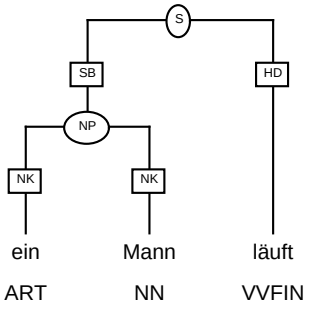


Abbildung 2.5: Ein einfacher Beispielsatz

Nach dem stand-off Ansatz wird der Korpustext in der Basisdatei festgehalten. Er ist sinnvollerweise durch Absätze und Satzgrenzen strukturiert:

```
<p><s id="s1">ein Mann läuft</s></p>
```

Die Wortarten-Annotation wird separat verwaltet. Man beachte, dass erst auf dieser sekundären Ebene die Einteilung des Textes in Token (repräsentiert durch `<tok>`-Elemente) durch die verwendeten XPointer erfolgt:

```
<p xlink:href="xptr(substring(//p[1]))">
  <s xlink:href="xptr(substring(//p[1]/s[1]))">
    <tok id="t1" xlink:href="substring(//p[1]/s[1]/text(),1,3)">
       <disamb>
          <pos>ART</pos>   <!-- ein -->
       </disamb>
    </tok>
    <tok id="t2" xlink:href="substring(//p[1]/s[1]/text(),5,4)">
       <disamb>
          <pos>NN</pos>    <!-- Mann -->
       </disamb>
    </tok>
    <tok id="t3" xlink:href="substring(//p[1]/s[1]/text(),10,5)">
       <disamb>
          <pos>VVFIN</pos> <!-- läuft -->
       </disamb>
    </tok>
  </s>
</p>
```

Der XPointer des ersten Tokens (d.h. des ersten `<tok>`-Elements) verweist beispielsweise auf die Zeichenkette *ein*, die beim ersten Buchstaben im ersten Satz im ersten Paragraphen beginnt und drei Buchstaben lang ist. Die Wortarten `<pos>` sind in einem Element `<disamb>` gekapselt, das andeuten soll, dass hier bereits eine Disambiguierung stattgefunden hat und eine zusätzliche Auflistung aller Lesarten sinnvoll sein könnte (vgl. Ide et al. 2000). Soll eine weitere Annotation auf Wortpositionsebene hinzugefügt werden (z.B. Lemma oder morphologische Markierung), so kann diese Annotation mit der Wortart zusammen als Geschwisterelement von `<pos>` abgelegt werden.

Die angegebene sekundäre Annotationsebene beschreibt eine eigene Sicht auf die Basisdaten. Sie bildet für sich ein eigenständiges, geschlossenes Korpus.

Die syntaktische Annotationsebene sieht nach den XCES-Empfehlungen folgermaßen aus: Da es im Beispielsatz keine Kantenkreuzungen gibt, können Konstituenten als sich einbettende `<struct>`-Elemente modelliert werden. Die an einer Konstituente abgelegten Attribut-Wert-Paare werden nicht als Attributwerte oder als Elementinhalt eines Elements abgelegt, das den Namen des Attributs trägt, da sonst eine Dokumentspezifikation durch DTD oder Schema zu stark korpusabhängig wäre. Stattdessen kommt ein Attributelement `<feat>` zum Einsatz. Die Relationen, die im Beispielsatz durch Kantenbeschriftungen ausgedrückt werden, werden durch explizite `<rel>`-Elemente kodiert.

Blattknoten des Baumes bleiben unspezifiziert, abgesehen vom Verweis auf den Basistext. Bei dieser Beispielmodellierung ist zu diskutieren, ob die Blätter aus linguistischer Sicht nicht besser an die Wortarten-Annotationsebene ange-

bunden werden sollten, die bereits eine Tokenisierung vornimmt. Denn eine Änderung der Tokenisierung hätte sonst Anpassungen an zwei Stellen zur Folge. Auch ist die Form der Referenzen zu überdenken, da die Paragraphen- und Satznummerierung Änderungen unterworfen sein könnte. Die Verwendung von Paragraphen- bzw. Satz-IDs scheint eine verlässlichere Variante.

```
<struct id="s0">

  <feat type="cat">S</feat>
  <rel type="SB" target="s1" />
  <rel type="HD" target="t2" />

  <struct id="s1">

    <feat type="cat">NP</feat>
    <rel type="NK" target="t0" />
    <rel type="NK" target="t1" />

    <struct id="t0">  <!-- ein -->
      <seg target="xptr(substring(//p[1]/s[1]/text(),1,3))" />
    </struct>

    <struct id="t1">  <!-- Mann -->
      <seg target="xptr(substring(//p[1]/s[1]/text(),5,4))" />
    </struct>

  </struct>

  <struct id="t2">    <!-- läuft -->
    <seg target="xptr(substring(//p[1]/s[1]/text(),10,5))" />
  </struct>

</struct>
```

Diese Form der Kodierung ist recht intuitiv und einleuchtend. Soll jedoch ein Satz mit kreuzenden Kanten kodiert werden, so trägt die Einbettung als Modellierung der Teil-Ganzes-Beziehung nicht weit genug. Zwar wird die Anordnung der Terminale streng genommen bereits auf der Wortarten-Ebene abgedeckt und eine Einbettung hätte nicht den Verlust der Tokenanordnung zur Folge. Doch eine Einbettung würde eine Anordnung der inneren Knoten modellieren, die hier nicht gegeben ist.

In diesem Falle empfehlen die XCES-Richtlinien, alle Knoten als Geschwisterelemente anzugeben, wodurch sich die Satzstruktur durch entsprechende

Referenzen ergibt (vgl. Ide und Romary 2001). Das folgende Beispiel zeigt die Kodierung des Satzes *Ein Mann kommt, der lacht* (vgl. Abb. Abb. 2.3 auf S. 19):

```
<struct id="sentence_id">

  <struct id="s0">
    <feat type="cat">S</feat>
    <rel type="SB" target="s1" />
    <rel type="HD" target="t2" />
  </struct>
  <struct id="s1">
    <feat type="cat">NP</feat>
    <rel type="NK" target="t0" />
    <rel type="NK" target="t1" />
    <rel type="RC" target="s2" />
  </struct>
  <struct id="s2">
    <feat type="cat">S</feat>
    <rel type="SB" target="t3" />
    <rel type="HD" target="t4" />
  </struct>

  <struct id="t0">  <!-- Ein -->
    <seg target="xptr(substring(/p/s[1]/text(),1,3))" />
  </struct>

  <struct id="t1">  <!-- Mann -->
    <seg target="xptr(substring(/p/s[1]/text(),5,4))" />
  </struct>

  <struct id="t2">  <!-- kommt -->
    <seg target="xptr(substring(/p/s[1]/text(),10,5))" />
  </struct>

  <struct id="t3">  <!-- der -->
    <seg target="xptr(substring(/p/s[1]/text(),17,3))" />
  </struct>

  <struct id="t4">  <!-- lacht -->
    <seg target="xptr(substring(/p/s[1]/text(),21,5))" />
  </struct>

</struct>
```

Diskussion

Der beschriebene XCES-Ansatz zur Standardisierung syntaktischer Annotation ist mächtig genug, die meisten Ansätze zur syntaktischen Annotation zu integrieren. Dazu kommen die Flexibilität des stand-off Ansatzes und die nicht zu unterschätzenden Vorteile der XML-Verwendung. Damit hat die XCES-konforme Kodierung von Baumbanken in jedem Falle einen hohen Stellenwert und sollte vom Suchwerkzeug unterstützt werden.

Doch eignet sich dieser Ansatz auch als Korpuseingangsformat? Bei der Beantwortung dieser Frage sollten einige Probleme bedacht werden:

- Die Mächtigkeit der Kodierung syntaktischer Annotation stellt zugleich einen Nachteil dar: Baumbanken wie die Penn Treebank werden sinnvollerweise den Einbettungsmechanismus verwenden, Baumbanken wie die TIGER-Baumbank hingegen den Referenzmechanismus. Auch ein Mischen der Ansätze ist zulässig. Die Korpusaufbereitungskomponente des Suchwerkzeugs muss demnach alle denkbaren Kodierungsstrategien unterstützen. Auch verarbeitende XSLT-Stylesheets müssen stets beide Möglichkeiten der Graph-Traversierung in Betracht ziehen.

  Für die eindeutige Repräsentation der Baumbanken durch das Suchwerkzeug sollte daher der Referenzmechanismus verwendet werden. Damit wird aber nur noch ein kleiner Teil der XCES-konformen Korpora direkt unterstüzt.

- Ähnliche Probleme entstehen durch die Flexibilität der stand-off Annotation des XCES-Ansatzes. Es bleibt jeder Baumbank selbst überlassen, wie die Attribut-Wert-Paare an den inneren und äußeren Knoten repräsentiert werden. Ein zuverlässiges Eingangsformat müsste also einen Standard über dem XCES-Standard definieren. Allerdings ist dann zu überlegen, ob nicht ein separates, auf das zu unterstützende Datenmodell zugeschnittenes Repräsentationsformat seinen Zweck besser erfüllt.

Insgesamt geben die technischen Probleme den Ausschlag, die XCES-Kodierung syntaktischer Annotation nicht als Eingangsformat zu verwenden. Um der Bedeutung des XCES-Ansatzes gerecht zu werden, sollte aber eine Unterstützung in Form eines Importfilters angeboten werden.

Für den Entwurf eines Eingangsformats haben sich einige wertvolle Hinweise ergeben. Zum einen hat die Verwendung von XML als Basisformat für eine Korpuskodierung entscheidende Vorteile, die sie gegenüber einem Format wie dem Negra-Format oder der Klammerstruktur-Darstellung überlegen machen. Auch sollte der Wiederverwendbarkeit eines Korpus ein hoher Stellenwert beigemessen werden.

2.4 Annotation graphs

Während sich der XCES-Ansatz darauf konzentriert, ein für alle Korpora einheitliches Datenformat zu etablieren, geht der Ansatz der *annotation graphs* einen anderen Weg. Dieser Ansatz entwickelt eine Daten*struktur*, die mächtig genug ist, möglichst viele Korpustypen (Textkorpora, Sprachkorpora) repräsentieren zu können (Bird und Liberman 2001). Der Vorteil einer solchen Vorgehensweise liegt auf der Hand: Verwenden alle Korpora dieselbe Datenstruktur, so können weiterverarbeitende Werkzeuge wie Suchwerkzeuge oder Visualisierungswerkzeuge direkt auf dieser einheitlichen Datenstruktur aufsetzen. Auf diese Weise wird eine größtmögliche Wiederverwendbarkeit von Werkzeugen erreicht. Eine Übersicht über aktuell verfügbare Werkzeuge ist in Bird et al. (2002) zu finden.

Die zentrale Frage dieser Idee besteht natürlich darin, ob es auch wirklich gelingt, die sehr unterschiedlichen Anforderungen der Korpora auf einen gemeinsamen Nenner zu bringen. An dieser Stelle wird gezeigt, wie das Datenmodell zur Kodierung syntaktischer Annotation eingesetzt werden kann. Die Vorgehensweise ist aus Cotton und Bird (2002) übernommen worden. Eine Umsetzung in weiteren linguistischen Bereichen wird in Bird und Liberman (2001) beschrieben.

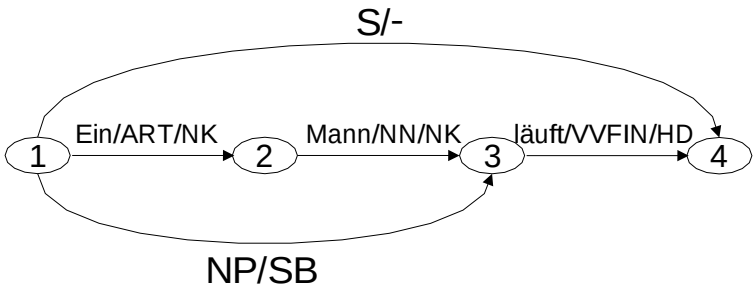


Abbildung 2.6: Ein Annotationsgraph für einen einfachen Beispielsatz

Zentraler Baustein des Ansatzes ist die Zeitachse, die die lineare Abfolge der Sprache repräsentiert (vgl. Abb. 2.6). Im Falle von Sprachkorpora werden die Sprachäußerungen dazu nicht nur in eine lineare Abfolge gebracht, sondern zusätzlich die genauen Start- und Endzeitpunkte der Teiläußerungen festgehalten (vgl. Bird und Liberman 2001).

Für die Kodierung einer syntaktischen Annotation wird die Zeitachse an der Tokenabfolge festgemacht. Sie wird modelliert als eine lineare Abfolge von Knoten, die nach der zeitlichen Abfolge angeordnet sind. Jeder Knoten ist mit einer ID versehen. Die Kantenübergänge tragen die eigentliche Token-Annotation, also im Beispiel die Wortform, Wortart und die eingehende Kantenbeschriftung.

Übergeordnete Kanten repräsentieren die Konstituenten. Sie umfassen als Annotation im Beispiel die syntaktische Kategorie sowie die eingehende Kantenbeschriftung. Da in den Wurzelknoten keine Kante führt, ist die fehlende

Beschriftung durch das Symbol "-" repräsentiert. Die Anordnung der Konstituenten richtet sich nach dem folgenden Prinzip (vgl. Bird und Liberman 2001, Abschnitt 3.3): Sind die von einer Kante w umspannten Token allesamt in den von einer Kante v umspannten Token enthalten, so umfasst die zu v gehörende Konstituente die Konstituente von w. Die exakte Konstituentenstruktur ergibt sich dann implizit aus den Beziehungen der Kanten zueinander. Im Beispiel umfasst die S-Konstituente die Nominalphrase und das finite Verb, während Artikel und Nomen zur Nominalphrase gehören.

Im Falle unärer Kanten und speziell im Falle von Kantenkreuzungen ist die Einbettung nicht eindeutig. Sie muss mit weiteren Hilfsmitteln ausgedrückt werden. Dazu werden jeder Kante zwei weitere Informationen zugeordnet (vgl. Abb. 2.7 und Abb. 2.8): Eine ID sorgt dafür, dass die Kanten eindeutig unterscheidbar sind. Ein Verweis auf den Elternknoten in Form seiner ID kodiert dann die Graphstruktur. Die Graphstruktur ist damit bottom-up kodiert.

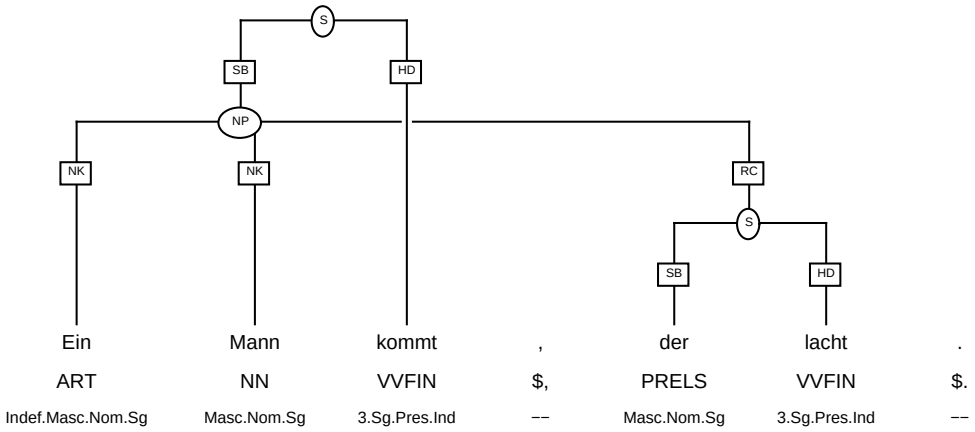


Abbildung 2.7: Ein Beispielsatz mit kreuzenden Kanten

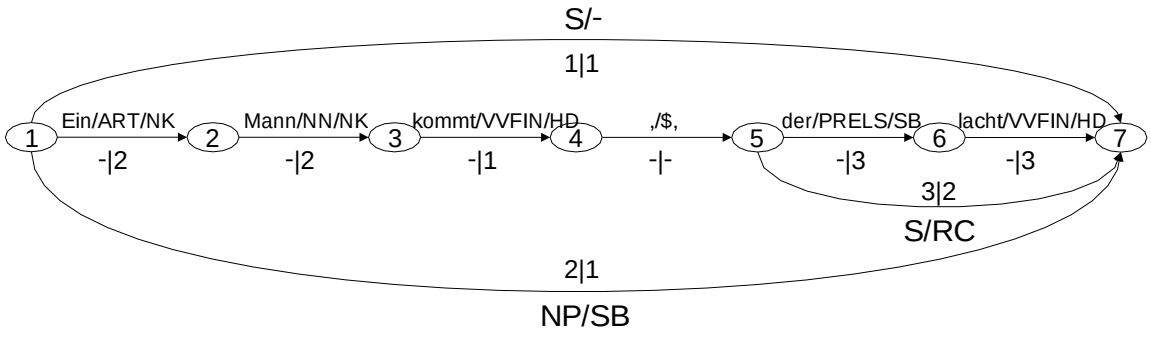


Abbildung 2.8: Der zugehörige Annotationsgraph

Die Vorteile des Ansatzes liegen auf der Hand:

- Setzt sich der Ansatz tatsächlich durch, d.h. unterstützen viele Projekte die vorgeschlagene Datenstruktur, so wird ein hoher Wiederverwendungsgrad der Daten und Werkzeuge erreicht.
- Durch eine bereits verfügbare XML-Kodierung für *annotation graphs* ist zugleich eine Standardisierung auf Kodierungsebene erreicht (vgl. Bird und Liberman 2001, Abschnitt 4.3).
- Auch ein speziell auf diese Datenstruktur zugeschnittenes Suchwerkzeug ist bereits in Vorbereitung (Bird et al. 2000).

Der beschriebene Ansatz stellt einen wesentlichen Beitrag für den Standardisierungsprozess innerhalb der Korpuslinguistik dar. Zur Kodierung von Baumbanken scheint der Ansatz aber nur bedingt geeignet:

- Die Mächtigkeit der Datenstruktur ist für Baumbanken überflüssig, da bis auf wenige Ausnahmen keine Zeitachse benötigt wird. Die Kodierung wird dadurch unnötig verkompliziert.
- Die Lesbarkeit einer Baumbankkodierung wird enorm erschwert, da die Strukturen nicht mehr erfassbar sind.
- Die beiden Beispiele zeigen lediglich, wie syntaktische Annotation in Form von Annotationsgraphen repräsentiert werden könnte. Um Baumbanken auf verlässliche Weise kodieren zu können, wird eine zusätzliche Standardisierung benötigt, die insbesondere die zugehörige XML-Repräsentation genau festlegt.

Ein Brückenschlag zwischen den Bereichen Baumbanken auf der einen Seite und Sprachkorpora auf der anderen Seite ist mit den *annotation graphs* sicher gelungen. Doch ist dieser Ansatz zu stark auf die Bedürfnisse der Sprachkorpora ausgerichtet, um auch im Bereich Baumbanken als zentrales Kodierungsformat agieren zu können. Aus diesem Grunde wird dieser Ansatz nicht als Korpuseingangsformat verwendet.

2.5 Zusammenfassung

In diesem Kapitel sind für die Kodierung von Baumbanken relevante Arbeiten vorgestellt worden. Für die Verwendung als Korpuseingangsformat ist keiner der Ansätze ohne Einschränkungen geeignet. Stattdessen wird im Laufe dieser Arbeit eine eigene Kodierung entwickelt, die auf das zu unterstützende Datenmodell zugeschnitten ist. Die zentralen Ideen der vorgestellten Arbeiten werden dabei maßgeblich in die Entwicklung des Formats einfließen. Sie haben gezeigt, dass:

- auch ein Baumbankkodierungsformat übersichtlich und lesbar sein kann,
- der Unterstützung bereits verfügbarer Baumbanken eine hohe Priorität eingeräumt werden muss,
- mit XML ein Basisformat zur Verfügung steht, das zur Kodierung von Textkorpora ideal geeignet ist.

Kapitel 3

Suchwerkzeuge für Baumbanken

Das vorliegende Kapitel hat die Aufgabe, bestehende Suchwerkzeuge für Baumbanken vorzustellen und zu diskutieren. Für jedes der Werkzeuge wird die Frage beantwortet, ob es als Suchwerkzeug im hier vorliegenden Kontext geeignet ist bzw. inwieweit es die formulierten Anforderungen erfüllt.

Interessant sind neben reinen Baumbank-Suchwerkzeugen auch solche Werkzeuge, die für die Suche auf der Wortebene entwickelt wurden und in Grenzen auch zur Suche auf Strukturen eingesetzt werden können. Stellvertretend für diese Arbeiten wird in Abschnitt 3.1 das Werkzeug *CQP* präsentiert.

Anschließend werden diejenigen Baumbank-Suchwerkzeuge vorgestellt, die bereits längere Zeit verfügbar und damit praxiserprobt sind. Es handelt sich um die Werkzeuge *tgrep*, *CorpusSearch*, *ICECUP*, *VIQTORYA* und *NetGraph*, die in den Abschnitten 3.2 bis 3.6 vorgestellt werden. Das *VIQTORYA*-System ist zwar noch nicht verfügbar, unterstützt aber das in dieser Arbeit verwendete Datenmodell und wird deshalb ebenfalls ausführlich behandelt.

Ein weiterer Themenkomplex in Abschnitt 3.7 befasst sich mit der Verwendung von XML-Suchwerkzeugen. Der anschließende Abschnitt 3.8 gibt einen Überblick über weitere Systeme, die weniger verbreitet oder noch nicht verfügbar sind. Der abschließende Abschnitt 3.9 fasst die wesentlichen Erkenntnisse dieses Kapitels zusammen.

3.1 CQP

Für Korpora, die auf Tokenebene annotiert worden sind (d.h. in denen jedem Token weitere linguistische Information wie Wortart oder Lemma zugeordnet ist), gibt es eine Fülle von Werkzeugen, die zum Teil auch kommerziell entwickelt werden. Für eine Übersicht sei auf Evert und Fitschen (2001) verwiesen. Zu den bekanntesten Vertretern gehören das für das British National Corpus entwickelte Programm *Sara* (Burnard 1996) und das zur IMS Corpus Workbench gehörende Programm *CQP* (Christ 1994, Christ et al. 1999). Stellvertre-

tend für die Arbeiten auf diesem Gebiet wird in diesem Abschnitt die Software CQP vorgestellt.

Der *Corpus Query Processor* CQP ist eine Software, die auch große Korpora (d.h. mit mehreren 100 Mio. Token) verarbeiten kann. Ein solches Korpus darf positionell (d.h. mit linguistischer Annotation am Token wie Wortart oder Lemma versehen) und strukturell (Satzgrenzen, Paragraphgrenzen usw.) annotiert sein. Um trotz der großen Datenmengen eine effiziente Anfrageverarbeitung sicherzustellen, muss ein Korpus zunächst einem Vorverarbeitungsschritt unterzogen werden, in dem ein so genannter Index angelegt wird. In diesem Index werden Daten über das Korpus gewonnen, die die spätere Anfrageverarbeitung beschleunigen. Da die meisten Korpora statisch sind, fällt der Aufwand für die Vorverarbeitung kaum ins Gewicht.

Ist ein Korpus aufbereitet, können seine positionellen Annotationen in Form von Attribut-Wert-Paaren abgefragt werden. In den folgenden Anfragebeispielen wird auf einem deutschen Gesetzestext-Korpus gesucht, das auf Tokenebene mit Wortart und Lemma annotiert ist. Die Wortart-Bezeichnungen richten sich dabei nach dem STTS-Tagset (vgl. Thielen et al. 1999). Die folgende Anfrage ermittelt alle Token, deren Wortform *Mann* lautet:

```
[word="Mann"];
```

Als Ausgabevisualisierung bietet CQP die Anzeige der Konkordanz, d.h. aller Korpusbelege für den Suchausdruck zusammen mit seinem linken und rechten Korpuskontext (*key word in context*). Der Kontext ist ebenso variabel einstellbar wie die Attribute, die hier angezeigt werden. Die Matches für die obige Anfrage sehen dann folgermaßen aus:

```
t er gewußt , daß der <Mann> oder die Frau verhei
widerrufen , wenn der <Mann> oder die Frau wahrhe
as Vermögen , das der <Mann> oder die Frau währen
das Gesamtgut von dem <Mann> oder der Frau oder v
Gesamtgutes durch den <Mann> oder die Frau # 4 §
Ehe empfangen und der <Mann> innerhalb der Empfän
rd vermutet , daß der <Mann> innerhalb der Empfän
mutung nur , wenn der <Mann> gestorben ist , ohne
s Kindes kann von dem <Mann> binnen zwei Jahren a
```

Um ein Token näher zu spezifizieren, können Attribut-Wert-Paare durch boolesche Ausdrücke miteinander verknüpft werden. Sie bilden ein sogenanntes Pattern, das in eckige Klammern eingeschlossen wird. Auf der Tokenebene sind reguläre Ausdrücke über Patterns zugelassen. Im folgenden Beispiel werden Phrasen gesucht, die aus einem optionalen Artikel (`ART`), beliebig vielen Adjektiven (`ADJA`) und dem Nomen *Mann* (`NN`) bestehen.

```
[pos="ART"]? [pos="ADJA"]* [pos="NN" & word="Mann"];
```

Mit Hilfe geeigneter Suchausdrücke kann nun versucht werden, linguistisches Wissen aus dem Korpus zu gewinnen. Der folgende Ausdruck ermittelt Kandidaten für Verb-Nomen-Kollokationen wie *(ein) Foto schießen*. Solche Kandidaten sind insbesondere für den Aufbau von Wörterbüchern interessant. Die Einschränkung des Suchkontextes auf einen Satz (durch `within 1 s`) verhindert, dass sich der Match über eine Satzgrenze hinaus ausdehnt. Mit der Wortart `VVFIN` wird ein finites Verb bezeichnet. Nach dem finiten Verb können beliebig viele Token folgen, die kein Nomen bezeichnen. Als Abschluss der gesuchten Struktur folgt ein Nomen.

```
[pos="VVFIN"] [pos!="NN"]* [pos="NN"] within 1 s;
```

Mit Hilfe einer Gruppierungsfunktion kann die Ausgabe auf die zu untersuchenden Verb-Nomen-Paare eingeschränkt werden. Nach Häufigkeit sortiert ergeben sich dann folgende Kollokationskandidaten:

Anwendung	finden	164
Vorschrift	finden	132
Zustimmung	bedürfen	36
Genehmigung	bedürfen	23
Erklärung	erfolgen	23
Zweifel	gelten	16

Neben der Satzgrenzenmarkierung können strukturelle Annotationen auch zur Kodierung syntaktischer Annotation eingesetzt werden (vgl. Evert 2001). Die folgende Anfrage untersucht Artikel-Nomen-Paare, die als Nominalphrase annotiert worden sind:

```
<np> [pos="ART"] [pos="NN"] </np>;
```

Die Grenzen der Abfragbarkeit syntaktisch annotierter Korpora bestehen in den Beschränkungen der Einbettung. CQP erlaubt keine Abfrage rekursiver Strukturen, d.h. beispielsweise von Nominalphrasen `<np>`, die wieder Nominalphrasen einbetten. Die Abfrage beschränkt sich auf maximale `<np>`-Annotationen. Als Alternativlösung werden eingebettete Nominalphrasen `<np>` bei der Indexierung automatisch in `<np1>` umbenannt:

```
<np> []* <np1> []* [pos="NN"] </np1> </np>;
```

Für viele Korpora ist diese Vorgehensweise durchaus praktikabel. Mit Hilfe von Makros kann sie weiter vereinfacht werden. Die Suche mit CQP ist zudem sehr effizient. Zur sinnvollen Abfrage tief annotierter Baumbanken reicht die Mächtigkeit der Anfragesprache jedoch nicht aus (vgl. Ide und Brew 2000).

Das Suchwerkzeug CQP ist mittlerweile zu einem Standardwerkzeug für Korpora geworden, die auf Tokenebene annotiert sind (vgl. Ide und Brew

2000). Die Anwendungsgebiete reichen von der Lexikographie (Docherty und Heid 1998, Heid et al. 2000) über die theoretische Linguistik (Meurers 2003) bis zum Einsatz bei der Annotation eines Textkorpus (Evert und Kermes 2002).

Es bleibt festzuhalten, dass Suchwerkzeuge wie CQP in Ansätzen auch zur Suche auf Baumbanken eingesetzt werden können. Da sie sehr große Korpora effizient verarbeiten, können sie daher zumindest als Ergänzung zu einem Baumbank-Suchwerkzeug verwendet werden. Ersetzen können sie die speziell entwickelten Werkzeuge jedoch nicht.

3.2 tgrep

Das bekannteste Baumbank-Suchwerkzeug ist *tgrep*, dessen Name bereits andeutet, dass das Werkzeug eine Mustersuche innerhalb einer Baumbank erlaubt (vgl. Pito 1993). Seinen hohen Bekanntheitsgrad verdankt das Werkzeug seiner Veröffentlichung mit der ersten Version der Penn Treebank im Jahre 1992 (vgl. auch Abschnitt 1.1).

Das Werkzeug tgrep ist auf die Bedürfnisse der Penn Treebank zugeschnitten. Es unterstützt ausschließlich das erweiterte Klammerstrukturformat und ist damit auf die Verarbeitung von Baumbanken beschränkt, die Baumstrukturen zur syntaktischen Annotation verwenden. Zwar könnte für Korpora wie die TIGER-Baumbank eine zusätzliche Kodierung der Tokenabfolge in das Klammerstrukturformat eingefügt werden (vgl. Ausführungen in Abschnitt 2.2), doch können diese Abfolgenummern in der Anfragesprache nicht miteinander in Beziehung gesetzt werden.

Systemarchitektur

Die Systemarchitektur sieht eine Zweiteilung vor (vgl. Abb. 3.1). In einem Vorverarbeitungsschritt wird ein zu durchsuchendes Korpus zunächst indexiert. Bei diesem Vorgang wird die Klammerstrukturdarstellung des Korpus eingelesen und in eine binäre Datei umgewandelt. Diese Binärdatei stellt dann die Datengrundlage für die Auswertung einer Suchanfrage dar. Die Laufzeit und der Speicherbedarf der Indexierung wachsen linear mit der Größe des Korpus. Als Faustformel kann festgehalten werden, dass die Indexierung pro 1 Million Token etwa 3-4 Minuten dauert und 256 MB Hauptspeicher in Anspruch nimmt. Für sehr große Korpora stößt das System damit an seine Grenzen.

Da die Indexierung nur vergleichsweise selten durchgeführt werden muss, fallen die Indexierungskosten kaum ins Gewicht. Die zeit- und speicherintensive Indexierung entlastet zudem die Anfrageverarbeitung, die meist in wenigen Sekunden abgeschlossen ist und konstanten Hauptspeicherbedarf aufweist. Leider ist weder die Indexierungs- noch die Suchstrategie von tgrep dokumentiert.

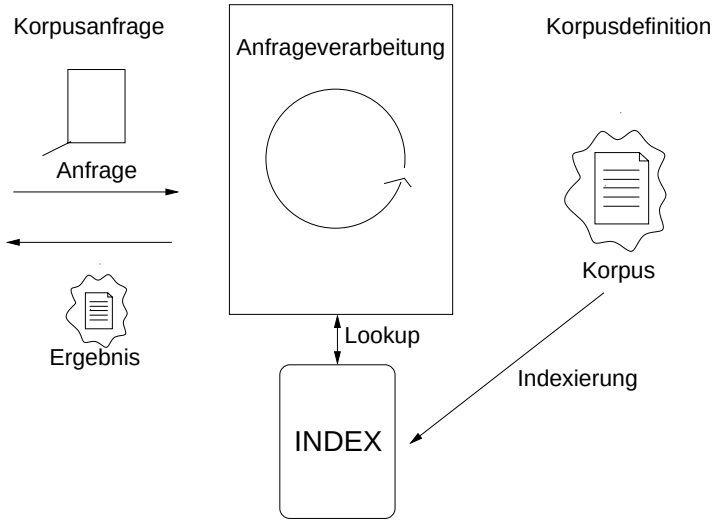


Abbildung 3.1: Systemarchitektur des Anfragewerkzeugs *tgrep*

Anfragesprache

Da die Klammerstrukturdarstellung verschiedenartige Informationen wie syntaktische Kategorie und syntaktische Funktion einfach nebeneinander schreibt, muss die Anfragesprache Mechanismen zum Isolieren der Informationen bereitstellen. Die Verwendung regulärer Ausdrücke steht dabei im Zentrum. Der folgende Suchausdruck beschreibt eine Nominalphrase, die eine weitere Nominalphrase einbettet. Die Verwendung der regulären Ausdrücke ist notwendig, um auch diejenigen Nominalphrasen zu berücksichtigen, denen eine Funktion zugeordnet worden ist (z.B. `NP-HD`).

```
/^NP.*$/ < /^NP.*$/
```

Durch die Aneinanderreihung von Relationen wird die Identität eines Knotens implizit ausgedrückt. Der folgende Ausdruck beschreibt eine Nominalphrase, die sowohl einen Artikel als auch ein Adjektiv direkt einbettet.

```
/^NP.*$/ < ART < ADJA
```

Durch entsprechende Klammerungen wird die Semantik so verändert, dass der zuletzt angegebene Knoten vom zweiten Knoten statt durch den ersten eingebettet wird:

```
/^NP.*$/ < (/^NP.*$/ < NN)
```

Um Kongruenzen ausdrücken zu können, wird der Registermechanismus regulärer Ausdrücke verwendet. In der folgenden Anfrage wird eine Nominalphrase beschrieben, die zwei nachfolgende Knoten mit der selben Kategorie umfasst.

Dazu wird die Kategorie des einen Knotens in runde Klammern eingeschlossen und als Kategorie des zweiten Knotens durch `\1` wieder aufgegriffen.

```
/^NP.*$/ < /^\(..*\)$/ < /\1/
```

Wie diese kurze Übersicht über die Anfragesprache zeigt, ist das Arbeiten mit tgrep gewöhnungsbedürftig. Die Verwendung regulärer Ausdrücke macht die Suchanfragen sehr unübersichtlich. Es fehlen Möglichkeiten der Modularisierung, die zur Wahrung der Übersicht beitragen würden.

Für größere Arbeiten steht allerdings ein leistungsfähiger Makroprozessor zur Verfügung, der auch kaskadenartige Anfragen zulässt. Dadurch lässt sich die Komplexität der Anfragen entzerren. Der Sprachumfang ist insgesamt ausreichend, auch wenn Ausdrucksmittel wie die Stelligkeit einer Konstituente fehlen. Aus linguistischer Sicht verwirrend ist die Definition der Präzedenzrelation, die an der Baumtraversierung festgemacht wird.

Diskussion

Insgesamt lassen sich folgende Vorzüge von tgrep festhalten:

- Das Werkzeug ist ausgereift und effizient.
- Mit dem mitgelieferten Makrotool steht ein mächtiges Instrument zur Verfügung, das nicht nur Suchanfragen, sondern auch Korpusmanipulationen auf der Grundlage von Anfrageergebnissen erlaubt.

Die Gewichtung der Nachteile von tgrep hängt stark vom Anwendertyp ab. Für gelegentliche Nutzer fällt die Bedienung stark ins Gewicht:

- Die Anfragesprache ist durch die Verwendung regulärer Ausdrücke unübersichtlich und schwer erlernbar.
- Es fehlt eine Dokumentation der formalen Semantik der Anfragesprache sowie eine Dokumentation des Konzepts der Anfrageverarbeitung.
- Durch die Einschränkung auf das Klammerstrukturformat müssen die Einzelinformationen mühsam isoliert werden. Zudem werden Korpora ausgeschlossen, die Graphstrukturen zur Annotation verwenden.
- Eine binäre Distribution von tgrep ist nur für die Solaris-Plattform verfügbar.

Da tgrep das zu unterstützende Datenmodell nicht verarbeiten kann, kann es in der vorliegenden Arbeit nicht als Suchwerkzeug verwendet werden.

tgrep2

Mittlerweile ist das tgrep-Werkzeug von Douglas Rohde unabhängig vom Penn Treebank Projekt reimplementiert worden (vgl. Rohde 2001). Die enstandene Software wird als *tgrep2* bezeichnet und ist in binärer Form für die Linux-Plattform verfügbar. Zu den wichtigsten Ergänzungen zählen die Einführung der Disjunktion, die Benennung von Knoten im Sinne einer Variablenverwendung und die Einführung eines eingeschränkten Allquantors. Die Semantik des Allquantors ist allerdings unklar und unzureichend dokumentiert. Zwar sind mit tgrep2 einige Bedienungsprobleme behoben worden, doch bleiben die konzeptionellen Schwächen bestehen.

3.3 CorpusSearch

Als Alternative zu Suchwerkzeugen wie tgrep, die auf die Verwendung regulärer Ausdrücke setzen, ist *CorpusSearch* entwickelt worden (vgl. Randall 2000). Das Hauptanliegen von CorpusSearch besteht darin, eine sprechende und leicht erlernbare Anfragesprache anzubieten, die auch vom Gelegenheitsanwender beherrscht werden kann. Das Werkzeug ist speziell für die Suche auf dem Penn-Helsinki Parsed Corpus of Middle English entwickelt worden. Da dieses Korpus eine Variante der Penn Treebank Annotation verwendet, ist CorpusSearch für alle auf dem erweiterten Klammerstrukturformat basierenden Baumbanken geeignet. Damit gibt es wie bei tgrep keine Möglichkeit, Graphenannotationen zu verarbeiten. Da keine Vergleichsoperatoren zur Verfügung stehen, hilft auch die numerische Kodierung der Tokenabfolge (vgl. Abschnitt 2.2) nicht weiter.

Systemarchitektur

Die Systemarchitektur sieht im Falle von CorpusSearch vor, die Anfrageverarbeitung stets auf dem Originalkorpus (d.h. auf der textuellen Klammerstrukturdarstellung) durchzuführen. Das Korpus wird also für jede Anfrage eingelesen, geparst und Satz für Satz verarbeitet (vgl. Abb. 3.2). Nachteil dieser Vorgehensweise ist natürlich der immer wiederkehrende Aufwand für das Einlesen des Korpus. Vorteil ist jedoch, dass stets auf der aktuellen Korpusvariante gesucht wird. Da die Anfrageergebnisse wieder im Klammerstrukturformat abgelegt werden, kann die Suche auch auf der Ergebnismenge fortgesetzt werden. Ideal ist diese Architektur in der Aufbauphase einer Baumbank, da jede Änderung sofort berücksichtigt wird. Die genaue Funktionsweise der satzweisen Auswertung der Anfrage ist leider nicht dokumentiert.

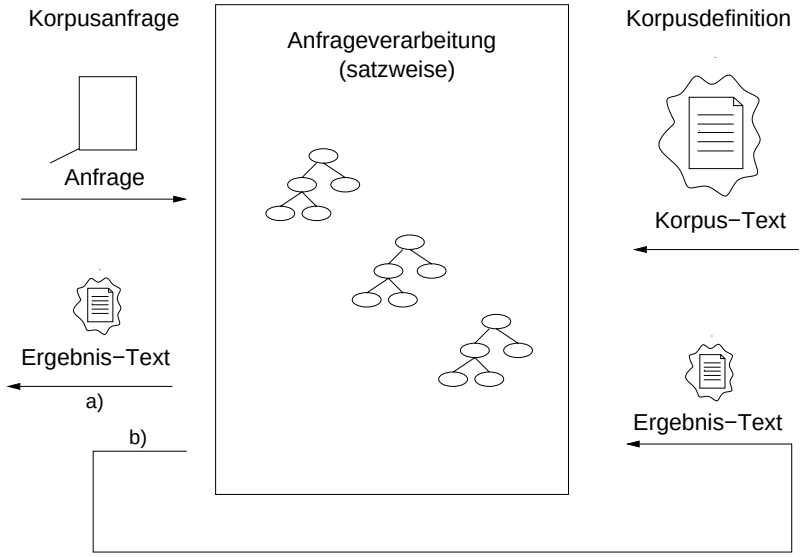


Abbildung 3.2: Systemarchitektur des Suchwerkzeugs *CorpusSearch*

Anfragesprache

Die Anfragesprache von CorpusSearch ist sehr mächtig und wirkt durchdacht. Zwar müssen durch die Fokussierung der Klammerstrukturdarstellung verschmolzene Informationen isoliert werden, doch geschieht dies mit Hilfe des Platzhaltersymbols `*` auf eine intuitive Art und Weise. Die folgende Anfrage beschreibt eine Nominalphrase mit einer beliebigen zugeordneten Funktion:

```
(NP* exists)
```

Die Verknüpfung von zwei Knoten wird durch Schlüsselwörter spezifiziert, die die Relation beschreiben. Die folgende Anfrage drückt eine direkte Dominanzbeziehung (`iDominates`) zwischen zwei Nominalphrasen aus. Soll die Dominanz nicht notwendigerweise direkt sein, lautet das Schlüsselwort entsprechend `dominates`.

```
(NP* iDominates NP*)
```

Um einen Teilgraphen zu spezifizieren, wird dessen Beschreibung aus einzelnen Relationen zusammengesetzt, die durch eine konjunktive Verknüpfung verbunden sind. Um denselben Knoten wieder aufgreifen zu können, wird eine Benennung durchgeführt. Das folgende Beispiel sucht nach einer Nominalphrase, die Artikel und Nomen umfasst:

```
([1]NP* iDominates ART) AND ([1]NP* iDominates NN)
```

Sehr hilfreich sind Knotenprädikate, mit dessen Hilfe Knoteneigenschaften in die Anfrage einbezogen werden können. Durch die Spezifikation der Stellig-

keit des Mutterknotens beispielsweise wird die Nominalphrase weiter eingeschränkt, eine Variante der Dominanzrelation spezifiziert dazu noch die Nummer des Nachfolgers und damit die Reihenfolge der Kinder. So ist der Artikel in der folgenden Anfrage stets der erste und das Nomen der zweite Nachfolger des Mutterknotens. Die Nominalphrase ist in eindeutiger Weise beschrieben:

```
([1]NP* iDomsNumber1 ART) AND ([1]NP* iDomsNumber2 NN) AND
                    ([1]NP* iDomsTotal2)
```

Diskussion

Als Vorzüge von CorpusSearch sind zu nennen:

- Die Anfragesprache ist durchdacht und ausdrucksstark. Sie ist dabei weder unübersichtlich noch schwer verständlich. Sie ist auch für den Gelegenheitsanwender geeignet.
- Die Dokumentation ist ausführlich und gibt auch dem weniger versierten Benutzer Hilfestellungen an die Hand.
- Durch die Suche auf den Originaldaten ist das Werkzeug für den Einsatz in Baumbankprojekten ideal geeignet. Auch kaskadenartige Anfragen werden unterstützt.
- CorpusSearch ist in Java implementiert worden und damit auf jeder Plattform lauffähig.

Dem gegenüber stehen einige konzeptionelle Nachteile:

- Durch das Einlesen des Korpus für jede neue Anfrage ist die Verarbeitungsgeschwindigkeit niedrig. Das vereinfachte Konzept lässt keine Filterstrategien zur Reduktion des Suchraums und damit zur Beschleunigung der Verarbeitung zu.
- Das Gesamtkonzept ist durch diese Einschränkungen klar umrissen und kann wenig flexibel an sich ändernde Anforderungen angepasst werden (fehlende Skalierbarkeit).
- Die Einschränkung auf das erweiterte Klammerstrukturformat schließt graphenannotierte Korpora aus.
- Für die Anfragesprache fehlt die Dokumentation der formalen Semantik.

Was dem gelungenen Werkzeug fehlt, ist eine grafische Benutzeroberfläche. Sie würde das Konzept, den Benutzerkreis durch eine intuitive Anfragesprache auch auf Gelegenheitsnutzer zu erweitern, unterstreichen und das Werkzeug

noch besser nutzbar machen. Insbesondere Benutzer, die an eine grafische Umgebung gewöhnt sind, werden sich mit der dateibasierten Vorgehensweise auf der Kommandozeile kaum zurechtfinden.

Für die vorliegende Arbeit gibt CorpusSearch einige wesentliche Impulse zur Gestaltung der Anfragesprache. Wegen der fehlenden Unterstützung von Graphenannotationen ist es aber nicht direkt einsetzbar.

3.4 ICECUP

Das Suchwerkzeug *ICECUP* ist im Rahmen der Initiative *International Corpus of English* (ICE) entstanden[1]. Innerhalb dieses Projekts werden seit 1990 englischsprachige Texte gesammelt und annotiert, um eine weltweite vergleichende Studie englischer Sprache durchzuführen (Greenbaum 1996, Nelson et al. 2002). Die regionalen Korpora bestehen aus jeweils 1 Million Token geschriebener und transkribierter gesprochener Sprache und werden für insgesamt 15 Regionen erfasst.

Konzept

Das Suchwerkzeug ICECUP (ICE Corpus Utiliy Program) ist mit dem Ziel entwickelt worden, die in der ICE-Initiative gesammelten Korpora zugänglich zu machen. Da hier Linguisten das Zielpublikum bilden, sollten die Daten auf eine Weise präsentiert werden, die einer datengetriebenen Arbeitsweise entgegenkommt.

Ein zentraler Gesichtspunkt bei der Konzeption von ICECUP ist die Kritik an logischen Anfragesprachen (vgl. Wallis und Nelson 2000). Hier wird argumentiert, dass logische Anfragesprachen schwer zu erlernen sind, da zweidimensionale Baumstrukturen durch eindimensionale Ausdrücke beschrieben werden müssen. Stattdessen wird eine rein visuelle Sicht auf ein Korpus vorgeschlagen.

Ein Korpus ist in einem solchen Konzept mit Hilfe eines Browsers einsehbar, der die Baumstrukturen visualisiert. Auf diese Weise kommt dieser Ansatz vor allem den Anwendern entgegen, die mit dem Annotationsschema eines Korpus noch nicht vertraut sind und sich zunächst einen Überblick über die Annotation verschaffen möchten. Doch auch die Anfragen an ein Korpus werden nach diesem Konzept visuell durch das Zeichnen von Baumfragmenten formuliert. Die Anfrageverarbeitung wertet die Zeichnung aus und sucht nach Korpusbelegen, die das skizzierte Muster enthalten. Damit bleiben Anfrageformulierung und Ergebnisvisualisierung stets auf einer gemeinsamen Ebene.

[1] Die Beschreibung und Diskussion von ICECUP bezieht sich auf die Version 3.0.

Systemarchitektur

Die zweigeteilte Systemarchitektur des ICECUP-Systems ähnelt der Architektur von *tgrep*: In einer Indexierungsphase wird das Korpus in eine binäre Darstellung gebracht, die die Grundlage für die Suche bildet. Die Indexierungskomponente wird dem Endanwender nicht zur Verfügung gestellt. Der Index wird in Wallis und Nelson (2000) detailliert beschrieben und verfolgt das Ziel, im Falle einer exakten Suchspezifikation schnell an die matchenden Sätze zu kommen und im Falle einer ungenaueren Suche die Satzkandidaten zumindest einschränken zu können. Da es sich bei den ICE-Korpora um Korpora mit beschränkter Größe handelt, sind Indexierungsstrategien problemlos anwendbar und der daraus entstehende Speicherbedarf ist als sekundär einzustufen. Die verwendeten Indexierungsmethoden sind Standardmethoden des Information Retrieval und basieren im Wesentlichen auf der Idee der Korpusinvertierung.

Datenmodell

Das Datenmodell richtet sich nach den ICE-Konventionen. Unterstützt wird damit Baumannotation, die für jeden Knoten eine syntaktische Kategorie und eine Funktion angibt. Zudem müssen die Korpora vollständig disambiguiert sein. Graphannotationen können in ICECUP nicht verwendet werden. Ein typischer Satz aus dem ICE-Korpus und seine Visualisierung in ICECUP ist als Bildschirmabzug in Abb. 3.3 zu finden. Entgegen den sonstigen Gewohnheiten wird die Baumstruktur hier von links nach rechts angeordnet, wodurch die Baumwurzel links und die Token am rechten Rand untereinander angeordnet zu finden sind. Die Darstellung lässt sich in ICECUP aber beliebig anpassen.

Grafische Anfragen mit *Fuzzy Tree Fragments*

Zur Formulierung von Anfragen werden die so genannten *Fuzzy Tree Fragments* (FTF) verwendet. Wie der Begriff *fuzzy* bereits andeutet, werden hier Baumfragmente gezeichnet, die nicht notwendigerweise voll spezifiziert sein müssen. Das visuelle FTF-Konzept ist recht komplex und kann an dieser Stelle nur an einem Beispiel illustriert werden. Eine detaillierte Einführungen kann in Wallis und Nelson (2000, 2001) nachgelesen werden.

Ein typisches Anfrage-Fragment ist in Abb. 3.4 abgebildet. Die Ausrichtung des Baumes ist wieder von links nach rechts gewählt worden, kann aber beliebig geändert werden. In diesem Beispiel wird ein Nominalphrasentyp untersucht, der auch im Beispielsatz vorkommt (*this parallel deregulation*). Die Nominalphrase besteht hier aus einer Determinerphrase, einer Adjektivphrase und einem Nomen; zudem wird *this* als Determiner festgelegt.

Den Schaltknöpfen kommt eine besondere Bedeutung zu, da sie die Beziehungen zwischen den Knoten verändern können. Durch die durchgezogenen

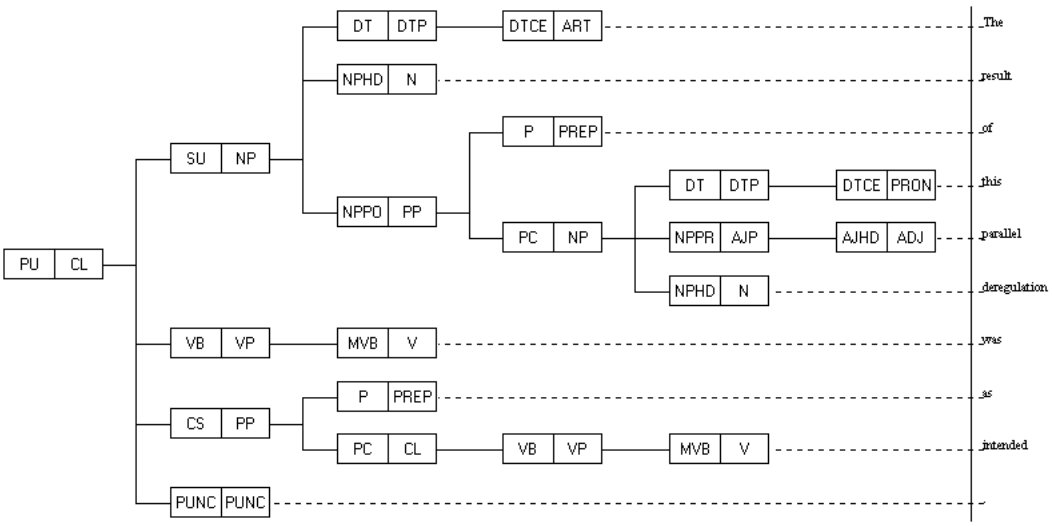


Abbildung 3.3: Bildschirmdarstellung des ICE-Beispielsatzes *The result of this parallel deregulation was as intended.*

Linien wird beispielsweise visualisiert, dass die untergeordneten Knoten direkte Nachfolger der Nominalphrase sind. Durch einen Klick auf den Knopf, der auf einer solchen Linie liegt, lässt sich diese Beziehung ändern, z.B. in eine unspezifizierte Dominanz. Die Pfeile deuten an, dass die drei Knoten direkte Nachbarn sind und kein Knoten mehr zwischen ihnen liegen darf. Auch hier kann durch einen Knopfdruck auf den Pfeil die Beziehung gewechselt werden, die sich wieder visuell niederschlägt. Die gestrichelte Linie zum Token *this* legt fest, dass das Token folgen muss, allerdings nicht direkt unter dem Knoten, sondern beliebig tief darunter.

Wie dieses Beispiel zeigt, lassen sich die FTF-Darstellungen zunächst leicht interpretieren, solange sich der Betrachter ganz auf die visuellen Elemente einlässt und die Schaltknöpfe ignoriert. Erst durch die Schaltknöpfe wirkt die Darstellung ein wenig überladen und unübersichtlich.

Bei der alltäglichen Arbeit mit dem Korpus finden die Anwender beim Sichten des Korpus häufig interessante Phänomene, die sie weiter verfolgen möchten. Hier kann der interessierende Subgraph markiert und automatisch in ein FTF konvertiert werden. Mit diesem mächtigen Mechanismus lässt sich in kürzester Zeit die gewünschte Suchanfrage starten.

Diskussion

Folgende Punkte machen ICECUP zu einem sehr gelungenen Suchwerkzeug:

- Das Werkzeug ist ausgereift und dank seiner professionellen Oberfläche sehr angenehm zu bedienen.

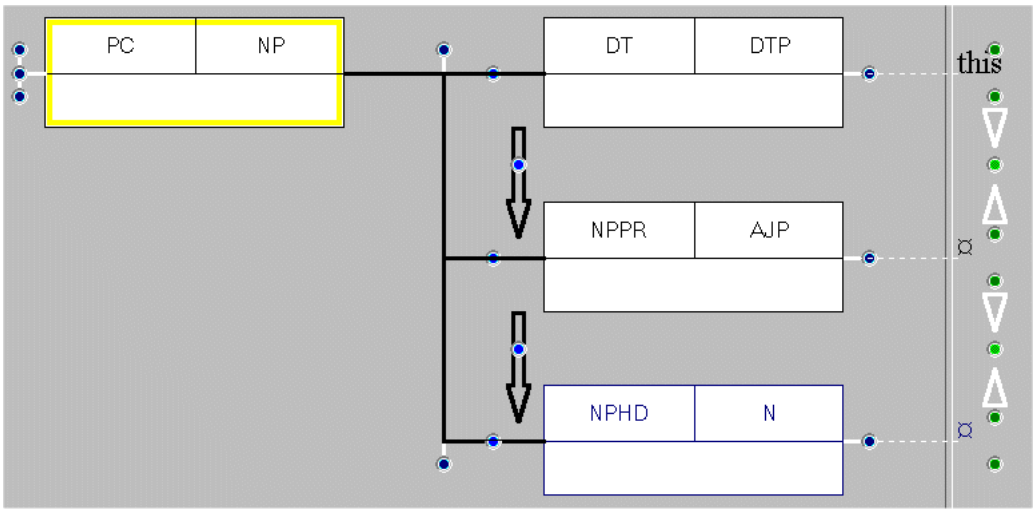


Abbildung 3.4: Ein Fuzzy Tree Fragment als Suchausdruck

- Das visuelle Konzept ist durchdacht und erlaubt eine intuitive Herangehensweise an ein Korpus.
- Die visuelle Anfragespezifikation schränkt die Ausdrucksmittel ein, wodurch eine effiziente Anfrageverarbeitung erfolgen kann.

Auf der anderen Seite gibt es einige Einschränkungen, die aber für den gedachten Einsatz des Werkzeugs im ICE-Projekt nicht relevant sind:

- Die Verarbeitung ist auf baumannotierte Baumbanken beschränkt.
- Die Ausdrucksmittel der grafischen Anfrage sind begrenzt, einige Ausdrucksmittel wie eine disjunktive Verknüpfung oder Negation atomarer Werte wären wünschenswert.
- ICECUP ist nur für die Windows-Plattform verfügbar.

Bedingt durch die fehlende Unterstützung des vorliegenden Datenmodells kommt ICECUP als Suchlösung in dieser Arbeit nicht in Betracht. Doch zeigt dieses Werkzeug deutlich, welchen Stellenwert die Entwicklung einer ausgefeilten grafischen Benutzeroberfläche einnimmt. Erst sie macht ein Suchwerkzeug leicht bedienbar. Interessant ist auch das Konzept der grafischen Anfrageeingabe. Eine solche Eingabemöglichkeit sollte zumindest als Ergänzung zu einer logischen Anfragesprache in Betracht gezogen werden.

3.5 VIQTORYA

Das VIQTORYA-System ist als Suchwerkzeug für die VerbMobil-Korpora entwickelt worden, die transkribierte Dialoge von Terminabsprachen enthalten (Hinrichs et al. 2000). Die Baumbank verwendet zur Annotation der syntaktischen Struktur eingeschränkte Graphstrukturen und greift dabei bis auf sekundäre Kanten auf das in dieser Arbeit verwendete Datenmodell zurück. Die Einheiten der VerbMobil-Baumbanken werden als so genannte *Turns* bezeichnet, d.h. ununterbrochene Äußerungen eines Dialogpartners (vgl. Beispiel in Abb. 3.5). Da VIQTORYA prinzipiell auch zur Suche auf anderen Korpustypen gedacht ist, wird im Folgenden der neutrale Begriff Korpusgraph verwendet.

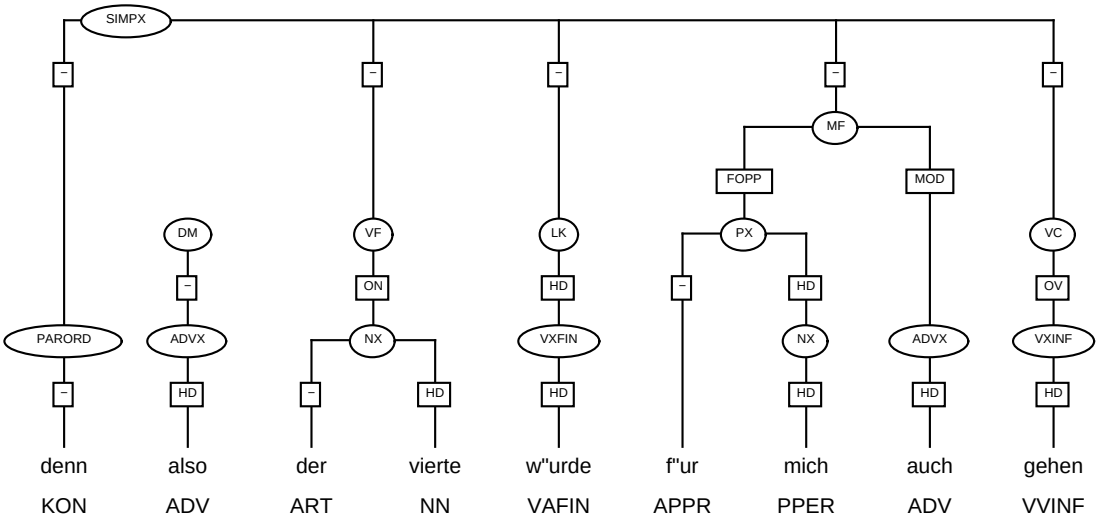


Abbildung 3.5: Ein Turn der VerbMobil-Baumbank

Systemarchitektur

Das Suchwerkzeug VIQTORYA sieht eine zweigeteilte Architektur vor (vgl. Kallmeyer 2000, Kallmeyer und Steiner 2002): In einem Vorverarbeitungsschritt wird das Korpus eingelesen und in eine relationale Datenbank überführt (vgl. Abb. 3.6). Die Datenbank dient als Datengrundlage für die Anfrageverarbeitung. Eine in einer logischen Anfragesprache formulierte Anfrage wird in einem abgetrennten Schritt in eine SQL-Anfrage übersetzt, die von der Datenbank ausgewertet wird. Da hier die Datenbank als Anfrageprozessor eingesetzt wird, entfällt die Entwicklung einer eigenen Anfrageverarbeitungs-Komponente. Es bleibt lediglich die Implementation der Anfrageübersetzung.

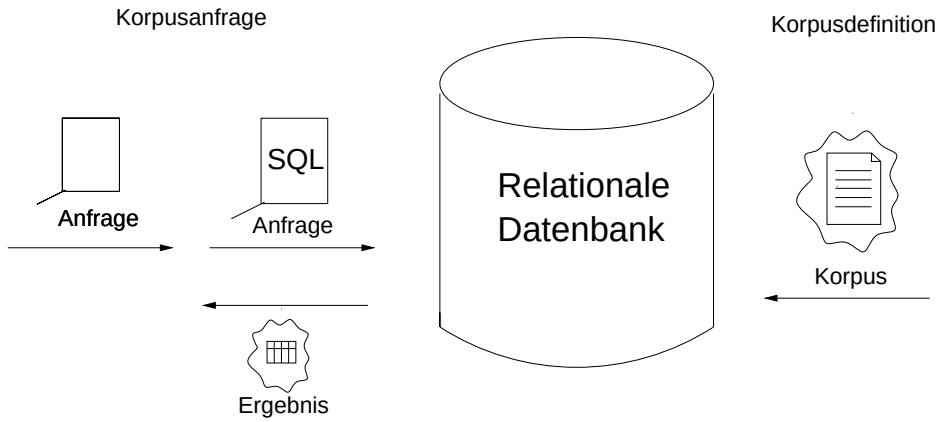


Abbildung 3.6: Systemarchitektur des Suchwerkzeugs *VIQTORYA*

Anfragesprache

Die Anfragesprache ist auf das Datenmodell und teilweise auch auf das VerbMobil-Korpus zugeschnitten. Für innere und äußere Knoten sind die beiden Attribute `cat` und `fct` definiert. Im Falle innerer Knoten bezeichnet `cat` die syntaktische Kategorie, andernfalls die Wortart. Das Attribut `fct` bezeichnet die durch die Kantenbeschriftungen ausgedrückten syntaktischen Funktionen. Eine Kantenbeschriftung wird demjenigen Knoten zugeordnet, auf den die Kante zeigt. Mit Hilfe des Attributs `token` kann die Wortform spezifiziert werden. Die folgende Anfrage beschreibt die Wortform *der* als Artikel:

```
cat(1)=ART & word(1)=der
```

Eine Korpusanfrage besteht aus einer Menge von Prädikaten und Relationen, die konjunktiv oder disjunktiv verknüpft werden können. Prädikate sind Attribut-Wert-Paare, die einen Knoten beschreiben. Relationen werden zur Spezifikation der Beziehung zweier Knoten eingesetzt. Um einen Knoten durch ein weiteres Prädikat oder eine Relation spezifizieren zu können, ordnet der Anwender einem Knoten eine Nummer als ID zu. Diese Nummer kann dann später wieder aufgegriffen werden. Im obigen Beispiel wird neben der Wortart auch die Wortform desselben Tokens spezifiziert. Neben der Gleichheit kann auch die Ungleichheit ausgedrückt werden. Das folgende Beispiel sucht nach Wortformen *der*, die nicht als Artikel verwendet werden.

```
cat(1)!=ART & word(1)=der
```

Als Relationen stehen die direkte Dominanz `>`, allgemeine Dominanz `>>` und (nicht unmittelbare) lineare Präzedenz `..` zur Verfügung. Alle Relationen können auch negiert werden. Das folgende Beispiel beschreibt eine Nominalphrase, die aus Artikel und Nomen besteht (vgl. Abb. 3.5):

```
cat(1)=NX & cat(2)=ART & cat(3)=NN & 1>2 & 1>3
```

Anfrageverarbeitung

Um eine Anfrage auf einem Korpus auswerten zu können, muss das Korpus zunächst in die Datenbank überführt werden. Im implementierten System wird die *mysql*-Datenbank benutzt. Durch die Verwendung der Programmiersprache Java und der Datenbank-Schnittstelle JDBC ist das System aber auf jede beliebige Datenbank anpassbar. Als Korpuseingangsformat wird das Negra-Format verwendet.

Das entwickelte Datenbank-Schema basiert auf der Beobachtung, dass die Aufzählung aller Knotenpaare eines Satzes zusammen mit ihren Beziehungen eine effiziente Prüfung von Knotenrelationen erlaubt. Dazu wird in der Vorverarbeitung für jedes Knotenpaar die Gültigkeit der Basisrelationen im Korpus abgelesen bzw. die Gültigkeit der erweiterten Relationen wie der allgemeinen Dominanz als transitive Hülle berechnet.

Zwar wächst der Speicherbedarf bei dieser Lösung quadratisch mit der Anzahl der Satzknoten, doch kann durch eine Klassenbildung für jedes Paar der Speicherbedarf auf wenige Bytes reduziert werden. Da die möglichen syntaktischen Kategorien und Funktionen ebenfalls beschränkt sind, werden sie zusätzlich in die Klassenbildung einbezogen. Allein die Wortformen müssen einzeln pro Token abgelegt werden.

Trotz der kompakten Klassenbildung darf der Platzbedarf nicht unterschätzt werden. Besteht ein Korpusgraph durchschnittlich aus 20 Token und 10 inneren Knoten und werden pro Paar nur vier Bytes zur Klassenbildung angesetzt, sind pro Graph immerhin 3.600 Bytes zzgl. Wortformenkodierung nötig. Für eine durchschnittliche Baumbank mit 50.000 Korpusgraphen ergeben sich bereits 180 MB Speicherbedarf ohne Wortformenkodierung. Um die Effizienz der Datenbank voll auszunutzen, muss zusätzlicher Speicherplatz für die Indexdaten der Datenbank eingeplant werden. Erste Zahlen für das VIQTORYA-System halten für die Verwaltung von etwa 10.000 Korpusgraphen Datendateien der Datenbank in Größe von 40 MB und Indexdateien in Größe von 400 MB fest (vgl. Kallmeyer 2000). Da der Speicherbedarf mit der Anzahl der Graphen wächst, könnte die Vorgehensweise damit für große Korpora unpraktikabel werden.

Eine Anfrage des Benutzers wird nun in eine SQL-Anfrage übersetzt und durch die Datenbank ausgewertet. Die Entwickler geben in Steiner und Kallmeyer (2002) die Auswertungsgeschwindigkeiten einiger Beispielanfragen an. Danach arbeitet das System effizient.

Falls die Anfrage disjunktive Verknüpfungen enthält, so wird vor der Transformation nach SQL eine Disjunktive Normalform der Anfrage erzeugt (vgl. Schurtz 2002). Als Ergebnis entsteht eine Liste von Disjunkten. Die Disjunkte werden nun unabhängig voneinander nach SQL transformiert und ausgewertet. Die Vereinigung der Anfrageergebnisse aller Disjunkte bildet dann das Anfrageergebnis der gesamten Anfrage. Die Laufzeit des Systems hängt durch diese Vorgehensweise unmittelbar von der Anzahl der erzeugten Disjunkte ab.

Benutzeroberfläche

Um das Suchwerkzeug leichter bedienbar zu machen und den Umgang mit der logischen Anfragesprache zu erleichtern, ist eine grafische Benutzeroberfläche entwickelt worden (vgl. Steiner und Kallmeyer 2002, Kallmeyer und Steiner 2002). Hier werden Knoteneigenschaften und Knotenrelationen per Mausklick festgelegt. Attributwerte können aus Attributwert-Listen ausgesucht werden, Relationstypen aus einer Liste der zu Verfügung stehenden Relationen. Abb. 3.7 zeigt, wie für eine Knotenrelation die beiden Knoten aus einer Liste bereits spezifizierter Knoten ausgewählt und mit der ausgewählten Dominanzbeziehung verknüpft werden. Diese Angaben werden anschließend in eine Relationsliste übertragen (vgl. Abb. 3.8). Aus allen Spezifikationen des Anwenders wird dann die entsprechende Anfrage generiert.

In der aktuellen Version ist die Benutzeroberfläche noch auf die VerbMobil-Korpora zugeschnitten, eine Verallgemeinerung ist aber geplant. Zur Visualisierung der Anfrageergebnisse kann das Annotationswerkzeug *Annotate* eingesetzt werden (vgl. Brants und Plaehn 2000), sofern die zu bearbeitenden Korpora zuvor in das Annotate-System eingespeist wurden.

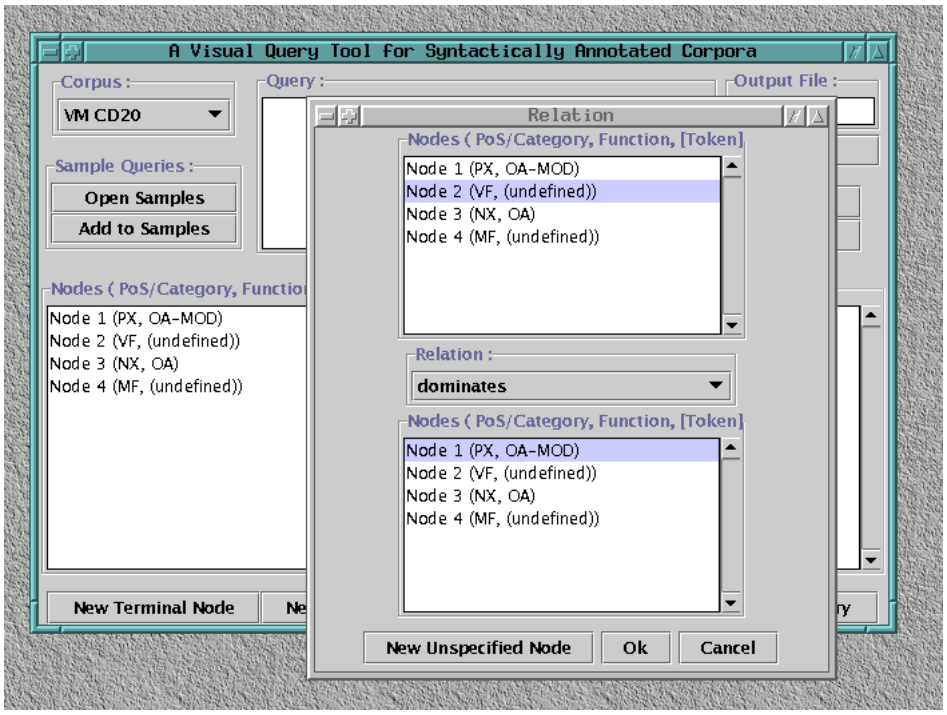


Abbildung 3.7: Grafische Spezifikation einer Knotenrelation

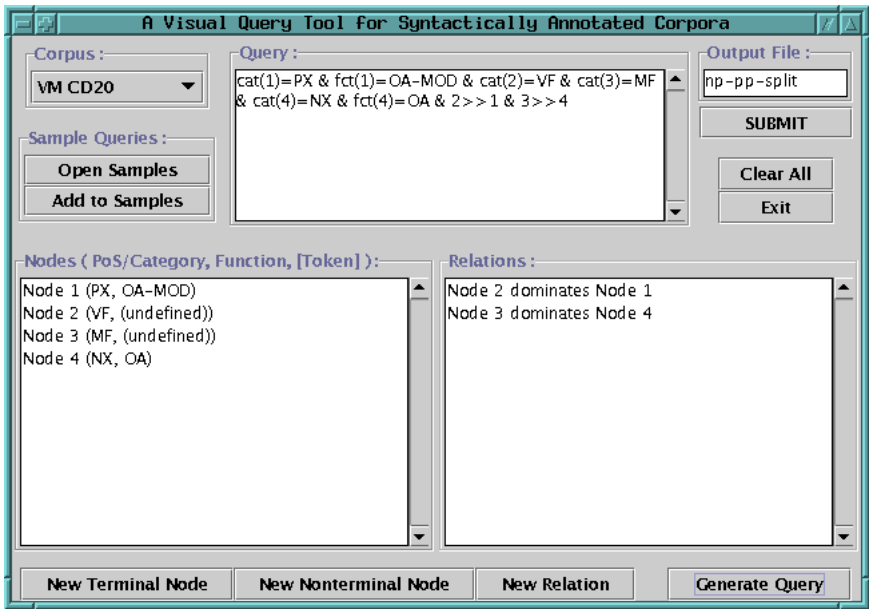


Abbildung 3.8: Knoten- und Relationsspezifikation und die zugehörige Anfrage

Diskussion

Die Vorzüge des VIQTORYA-Systems sind folgende:

- Die Anfrageverarbeitung wird mit Hilfe einer Datenbanklösung realisiert. Damit wird eine ausgereifte und effiziente Technologie eingesetzt.
- Durch diese Konzeption wird die Implementation einer eigenen Anfrageverarbeitung (mit Ausnahme der Übersetzung der Anfrage in SQL) überflüssig.
- Die Dokumentation der Anfragesprache umfasst auch eine formale Semantik.
- Eine Benutzeroberfläche erleichtert den Umgang mit der Anfragesprache.
- Durch die Verwendung der Programmiersprache Java ist die Software auf allen Plattformen einsetzbar.

Doch die Konzeption hat auch Nachteile:

- Eine Datenbanklösung erschwert die Distribution des Werkzeugs. Entweder muss beim Anwender die Verfügbarkeit einer Datenbank vorausgesetzt werden. Oder die Datenbank muss mit dem Werkzeug zusammen ausgeliefert und installiert werden.

- Der Speicherbedarf der Datenbanklösung ist sehr hoch. Für große Korpora könnte die Vorgehensweise unpraktikabel werden.
- Die Anfragesprache, das Korpusaufbereitungswerkzeug und die grafische Oberfläche sind auf das VerbMobil-Korpus bzw. Korpora mit analogen Eigenschaften zugeschnitten. Eine Verwendung eines anderen Korpustyps ist zur Zeit nicht möglich.
- Durch die Verwendung des Negra-Formats als Korpuseingangsformat bleibt das System zunächst auf einen fest vorgegebenen Korpustyp beschränkt.

Das VIQTORYA-System unterstützt als einziges System (abgesehen von sekundären Kanten) das geforderte Datenmodell. Doch insbesondere die Fokussierung auf das Negra-Format und die Abhängigkeit von der Datenbank machen das System bzgl. der für diese Arbeit formulierten Anforderungen zum gegenwärtigen Entwicklungsstand noch zu unflexibel.

3.6 NetGraph

Das Suchwerkzeug *NetGraph* ist speziell für die Suche auf der *Prague Dependency Treebank* (vgl. Hajic 1999) entwickelt worden. Eine Verarbeitung von Korpora mit einem anderen Datenmodell (z.B. TIGER-Baumbank) ist zwar denkbar (z.B. durch Kodierung der Präzedenz über ein Attribut). Wenn das NetGraph-Werkzeug aber als Gesamtheit gesehen wird (Anfragesprache, grafische Darstellung von Korpusgraphen usw.), macht diese Vorgehensweise wenig Sinn (vgl. auch Beispielsatz in Abb. 3.9). Dennoch ist die Konzeption und Architektur des Werkzeugs von Interesse, da hier vergleichbare Fragestellungen entwickelt und gelöst worden sind (vgl. Mírovský et al. 2002).

Anfragesprache

Das verwendete Datenmodell sieht gerichtete Graphen vor, in dem jeder Knoten (mit Ausnahme des Wurzelknotens) je ein Token repräsentiert. An jedem Knoten sind diverse Attribut-Wert-Paare für Attribute wie Wortform, Lemma, Wortart, morphologische Markierung usw. abgelegt. Ein weiteres Attribut kodiert die Position des Tokens im Satz als Nummer. Die Tokenabfolge wird grafisch durch die horizontale Anordnung der Knoten dargestellt (vgl. Abb. 3.9).

In der Anfragesprache werden Attribut-Wert-Paare zur Spezifikation der Knotenattribute aufgezählt und damit konjunktiv verknüpft. Im folgenden Beispiel wird nach dem Lemma *president* gesucht, dessen syntaktische Funktion als Subjekt markiert worden ist:

```
[lemma=president, afun=Sb]
```

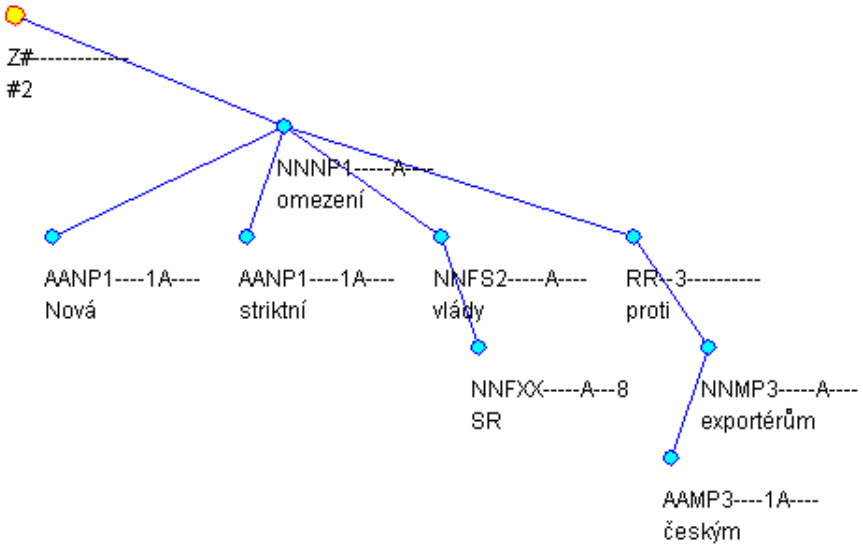


Abbildung 3.9: Ein Beispielsatz der Prague Dependency Treebank

Zur Unterspezifikation sind der Disjunktionsoperator | sowie die Symbole . und * vorgesehen, die einen beliebigen Buchstaben bzw. eine beliebige Zeichenkette repräsentieren. Im folgenden Beispiel muss das Lemma lediglich mit der Zeichenkette *pres* beginnen, die Funktion kann ein Subjekt oder ein Objekt sein:

```
[lemma=pres*, afun=Sb|Obj]
```

Um Knoten in ihrer Baumstruktur abfragen zu können, wird die Baumstruktur linearisiert. Jedem Knoten wird dazu eine Liste seiner Teilbäume – eingeschlossen in runde Klammern und getrennt durch Kommata – zugeordnet. Das folgende Beispiel zeigt die Beschreibung eines Baums mit insgesamt fünf Knoten (vgl. auch grafische Visualisierung in Abb. 3.10):

```
[tag=V*]([tag=P4*],[]([tag=V*]([tag=P4*])))
```

Um weitere Unterspezifikationen wie nicht-unmittelbare Einbettung eines Knotens oder auch genauere Spezifikationen wie die Anzahl der eingebetteten Kinder eines Knotens ausdrücken zu können, werden so genannte Meta-Attribute verwendet. Sie ergänzen die Spezifikation der Attribut-Wert-Paare zur Beschreibung eines Knotens. Im folgenden Beispiel muss der erste Knoten genau zwei Töchterknoten umfassen (`_#sons=2`), zudem darf der zweite Knoten beliebig tief eingebettet sein (`_transitive=true`):

```
[_#sons=2]([afun=Sb,_transitive=true])
```

Benutzeroberfläche

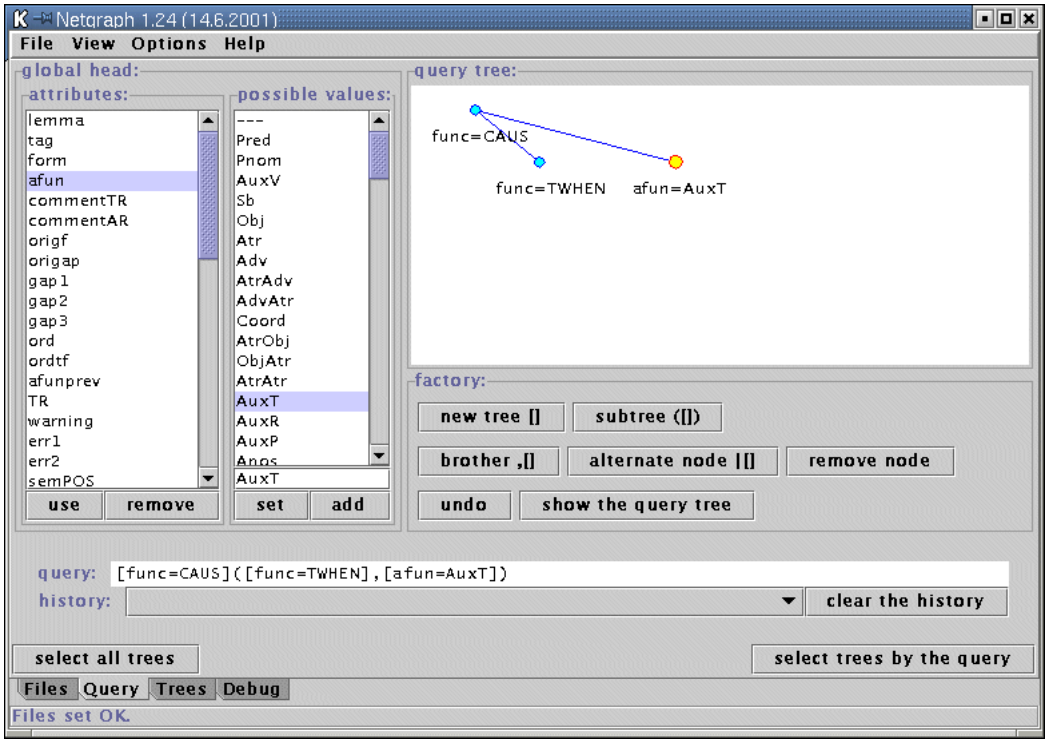


Abbildung 3.10: Formulierung von Suchanfragen

Die Benutzeroberfläche von NetGraph präsentiert sich sehr ansprechend. Eine Anfrage kann sowohl textuell eingegeben als auch über eine Eingabehilfe per Mausklick aufgebaut werden (Abb. 3.10). Alle verfügbaren Attribute und ihre möglichen Belegungen werden ebenfalls zur Auswahl angeboten. Sehr hilfreich ist die Visualisierung der Anfrage in Form einer Baumdarstellung. Damit bleibt die gesuchte Struktur selbst bei komplexen Anfragen transparent.

Ist die Suche erfolgreich, so werden alle matchenden Korpussätze in einer grafischen Darstellung angezeigt (vgl. Abb. 3.11). Die am Match beteiligten Knoten werden hervorgehoben, die an den Knoten angezeigten Attribute sind auswählbar.

Bemerkenswert an der Systemarchitektur von Netgraph ist die konsequente Umsetzung einer Client/Server-Architektur. Der Anfrageprozessor ist auf einem zentralen Server installiert. Anfrage-Clients können sowohl aus einer lokalen Applikation als auch von einem Java-Applet aus gestartet werden. Durch diese Architektur bleiben die Clients schlank, die rechenintensive Auswertung von Suchanfragen bleibt dem leistungsstarken Server vorbehalten.

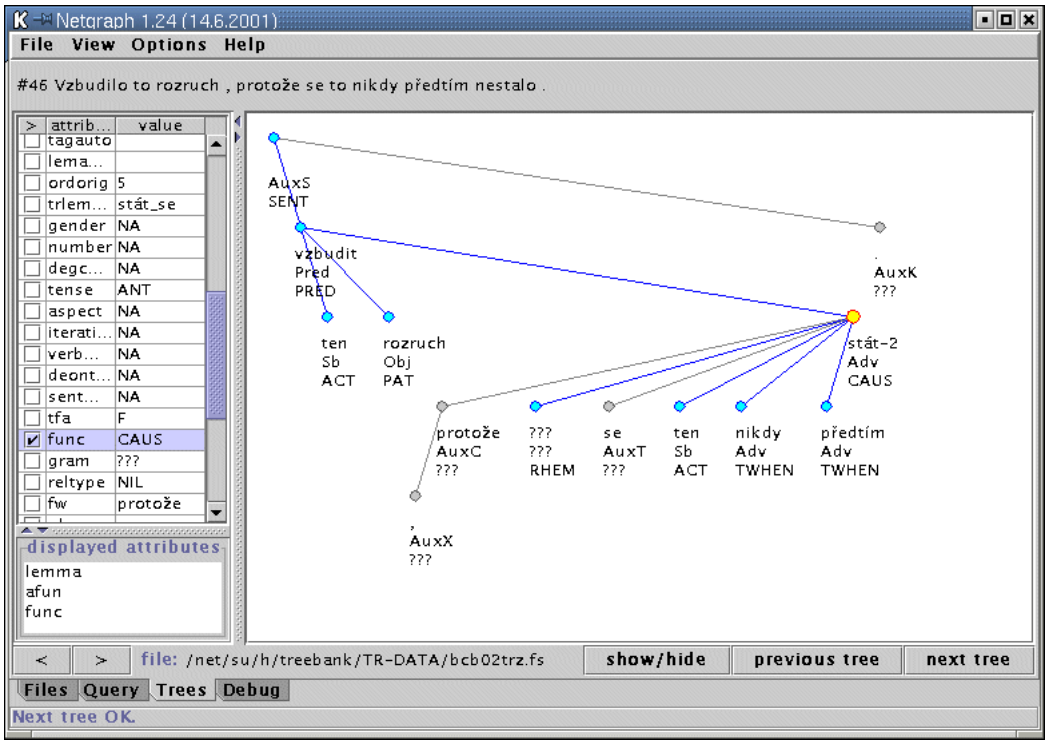


Abbildung 3.11: Grafische Visualisierung von Suchanfragen

Diskussion

Folgende Merkmale zeichnen Netgraph in besonderer Weise aus:

- Die Benutzeroberfläche ist durchdacht und erlaubt ein angenehmes Arbeiten. Die Visualisierung der Korpusbäume ist rundum gelungen.
- Das Client/Server-Szenario erlaubt den Zugriff auf das Korpus von jedem Browser aus.

Demgegenüber stehen folgende Nachteile:

- Die textuelle Anfragesprache wird durch die Klammerung von Einbettungen schnell unübersichtlich. Ohne eine Visualisierung sind Suchausdrücke nur schwer nachzuvollziehen.
- Die interne Verarbeitung der Suchanfragen ist nicht dokumentiert. Es fehlt ebenfalls eine formale Spezifikation der Anfragesprache.
- Das System ist auf die Verarbeitung eines Korpus beschränkt. Eine Aufbereitung eines neuen Korpus durch den Anwender ist nicht vorgesehen.

Das NetGraph-System macht insgesamt einen sehr ausgereiften Eindruck. Insbesondere die ansprechende Benutzeroberfläche ist eine hilfreiche Anregung für die Konzeption der Oberfläche des in dieser Arbeit entwickelten Suchwerkzeugs.

3.7 XML-Suchwerkzeuge

Für die Kodierung linguistisch annotierter Daten wird immer häufiger XML als Basisformat vorgeschlagen. Die Vorteile, die sich aus einer XML-Verwendung ergeben, sind im vorangegangenen Kapitel bereits deutlich geworden (vgl. Abschnitt 2.3). Doch warum spezialisierte Suchwerkzeuge für diese XML-Kodierungen entwickeln, wenn durch Forschungsprojekte und kommerzielle Initiativen Software entwickelt worden ist, die zur Suche auf XML-Dokumenten und damit unmittelbar zur Suche auf den XML-kodierten Korpora eingesetzt werden kann? Dieser Abschnitt beleuchtet, inwieweit diese Werkzeuge tatsächlich den Anforderungen dieser Arbeit gerecht werden. Die Darstellung erfolgt chronologisch.

An dieser Stelle erfolgt ein Vorgriff auf den zweiten Teil dieser Arbeit. In Kapitel 7 wird eine XML-Kodierung als Eingangsformat für das Suchwerkzeug entwickelt und begründet. Die Untersuchungen der XML-Werkzeuge wurden stets auf dieser Kodierung durchgeführt. Das folgende Beispiel zeigt die XML-Repräsentation des Satzes *ein Mann läuft.* (vgl. Abb. 3.12) und illustriert, wie Referenzen zur Kodierung der Baumstruktur eingesetzt werden.

```
<graph root="n2">
  <terminals>
    <t id="w1" word="ein"   pos="ART" />
    <t id="w2" word="Mann"  pos="NN" />
    <t id="w3" word="läuft" pos="VVFIN" />
  </terminals>

  <nonterminals>
    <nt id="n1" cat="NP">
      <edge idref="w1" />
      <edge idref="w2" />
    </nt>
    <nt id="n2" cat="S">
      <edge idref="n1" />
      <edge idref="w3" />
    </nt>
  </nonterminals>
</graph>
```

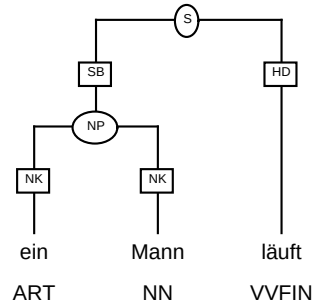


Abbildung 3.12: Ein Beispielsatz

Semistrukturierte Daten (1997)

Bevor XML in den Blickpunkt des Interesses rückte, konzentrierten sich Arbeiten, die sich mir der Suche auf hierarchisch organisierten Daten befassten, auf semistrukturierte Daten. Darunter werden gerichtete Graphstrukturen verstanden, die an den Kanten Beschriftungen und an den Knoten Werte elementarer Datentypen (Zeichenketten, ganze Zahlen, Gleitkommazahlen u.a.) tragen. In diesem recht allgemeinen Modell können beliebige hierarchische Informationstypen ausgedrückt werden, darunter auch Baumbanken.

Abb. 3.13 zeigt, wie der oben angegebene Beispielsatz in diesem Datenmodell kodiert werden könnte. Dabei sind zur besseren Lesbarkeit die Terminal- und Nichtterminalknoten des Korpusgraphen durch ein Rechteck hervorgehoben. Die zusätzlich als Ankerknoten für die Attribut-Wert-Paare eingefügten Hilfsknoten sind durch Ellipsen gekennzeichnet.

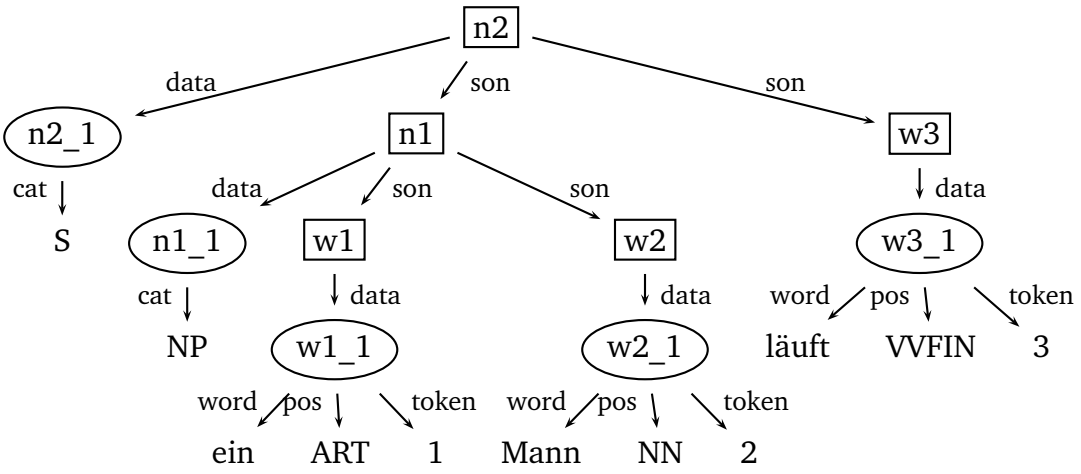


Abbildung 3.13: Der Beispielsatz als ungeordneter Graph

Für ein solch allgemeines Datenmodell sind umfassende Indexierungstechniken einzusetzen, damit die Verarbeitung von Anfragen in akzeptabler Zeit verläuft. Zu den bekanntesten implementierten Systeme in diesem Bereich zählt *Lore* (vgl. McHugh 2000). Die Anfragesprache von Lore orientiert sich an SQL und erlaubt die Abfrage beliebiger Beziehungen innerhalb des Graphen. Soll ein System wie Lore zur Suche auf Baumbanken eingesetzt werden, so sind zunächst Erweiterungen der Daten erforderlich. Da Lore auf einem ungeordneten Graphen arbeitet, muss die Präzedenzinformation in den Graphen eingetragen werden (vgl. zusätzliche numerische *token*-Information an den Wortknoten in Abb. 3.13).

Vorteil der Verwendung eines solchen Systems ist die Kostenersparnis, da die Entwicklung eines eigenen Systems überflüssig wird. Nachteile bestehen u.a. in der komplexen Anfragesprache, die unabhängig von späteren Anwendungen konzipiert wurde und auf den Baumbank-Anwender wenig intuitiv wirkt. Als Ausweg könnte eine linguistische Anfragesprache entwickelt werden, die auf die Anfragesprache des Systems abgebildet wird. Es bleiben technische Probleme wie die Einbindung in die eigene Implementation und die spätere schwierige Distribution und Installation des eigenen und des fremden Systems.

Da sich die Entwicklung im Bereich semistrukturierter Daten bereits zu einem frühen Zeitpunkt deutlich in Richtung XML bewegte (vgl. Goldman et al. 1999) und mit einer Korpuskodierung in XML deutliche Vorteile verbunden sind (vgl. Abschnitt 2.3), wurde in der Planungsphase der vorliegenden Arbeit entschieden, die neueren XML-basierten Entwicklungen zu verfolgen und nicht auf die schwer umzusetzende Einbettung eines Anfragesystems für semistrukturierte Daten zu setzen.

XQL (1998)

Die beiden ersten Spezifikationen für XML-Anfragesprachen und ihre Implementationen sind 1998 veröffentlicht worden. Mit XQL und XML-QL sind zwei Sprachen entwickelt worden, die zwar über eine ähnliche Ausdruckskraft verfügen, deren Syntax sich aber deutlich unterscheidet.

Die Anfragesprache *XQL* liegt als W3C Proposal vor (Robie et al. 1998). XQL ist eine pfadorientierte Anfragesprache, die bis auf wenige Einzelheiten mit den sogenannten *Patterns* der Transformationssprache XSLT übereinstimmt. Sie ähnelt der später veröffentlichten pfadorientierten Sprache *XPath*, ist aber längst nicht so mächtig. Der folgende Ausdruck zeigt einen typischen XQL-Pfadausdruck:

```
//graph/nonterminals/nt[cat='NP']
```

Der Pfad beschreibt ein `<graph>`-Element, das beliebig tief im XML-Dokumentbaum eingebettet ist und als direkten Nachfolger ein

<nonterminals>-Element und dieses wiederum als Nachfolger ein <nt>-Element umfasst. Das <nt>-Element muss als Wert des cat-Attributs NP besitzen. Die formulierte Anfrage würde also beispielsweise den Nominalphrasenknoten im oben gezeigten XML-Dokument finden.

Schwierig ist die Abfrage der Dominanzbeziehungen der Graphstruktur, da in XQL nur sehr eingeschränkte Möglichkeiten zum Verfolgen von Referenzen bestehen. Einzige Möglichkeit ist eine Abfrage der Übereinstimmung der idref- und id-Attribute. Im folgenden Beispiel wird nach einer Nominalphrase gesucht, die ein Nomen umfasst:

```
t[(@id = //nt[@cat='NP']/edge/@idref) and (@pos = 'NN')]
```

Erweiterungen der Sprache sehen zwar Funktionen zur einfacheren Referenzauswertung vor, die Beschränkung auf die Abfrage direkter Nachfolgebeziehungen bleibt aber bestehen.

Alternativ könnten die Dominanzbeziehungen als Teil-Ganzes-Beziehungen durch Einbettungen im XML-Dokument modelliert werden. Damit ließen sie sich mit Hilfe von Pfadausdrücken deutlich leichter beschreiben. Die dabei verlorengehende Anordnung der Token muss aber durch ein zusätzliches Positionsattribut aufgefangen werden, dessen Abfrage wieder ähnlich umständlich ist wie die Abfrage der Referenzen.

XML-QL (1998)

Die Spezifikation von *XML-QL* hebt die Einschränkungen von XQL weitgehend auf (vgl. Deutsch et al. 1998). Suchausdrücke in dieser Anfragesprache erinnern an SQL-Anfragen. Nützlich ist die Möglichkeit, eine veränderte Ausgabe des Anfrageergebnisses zu erzeugen. Die folgende Anfrage sucht nach einem <nt>-Element, das eine Nominalphrase beschreibt. Als Anfrageergebnis wird eine Liste von <match>-Elementen erzeugt, die die IDs matchender <nt>-Elemente enthalten.

```
construct <match>$id</match>
where     <graph>
            <nonterminals>
              <nt id=$id cat="NP"></>
            </nonterminals>
          </graph>
          in "test.xml"
```

Die Abfrage der Dominanz wird erleichtert durch eine explizite Unterstützung von IDREF-Referenzen. Liegt eine solche Referenz vor, so kann sie auf folgende Art und Weise beschrieben werden:

```
construct <match>$id</match>
where     <graph>
            <nonterminals>
              <nt cat="NP"><edge><idref id=$id pos="NN"></></></></>
            </nonterminals>
          </graph>
          in "test.xml"
```

Das IDREF-Attribut *idref* wird als eigenes Element verwendet, das auf den spezifizierten Nachfolger zeigt. Es steht also hier symbolisch für ein <t>-Element. Durch einen regulären Pfadausdruck wird aus der direkten Dominanz eine beliebige Dominanz:

```
construct <match>$id</match>
where     <graph>
            <nonterminals>
              <nt cat="NP"><(edge.idref)*><edge>
                           <idrefs id=$id pos="NN"></></></></></>
            </nonterminals>
          </graph>
          in "test.xml"
```

Eine genaue Stufung ist jedoch nicht spezifizierbar. Zum Vergleich der Anordnung zweier Elemente sind Ordnungsvariablen vorgesehen. Im folgenden Beispiel werden ein Artikel und ein Nomen beschrieben, wobei der Artikel vor dem Nomen zu finden ist.

```
construct <match><t1>$t1</t1><t2>$t2</t2></match>
where     <graph>
            <terminals>$t</terminals>
          </graph>
          in "test.xml",
          <t pos="ART" id=$t1></>[$pos1] in $t,
          <t pos="NN" id=$t2></>[$pos2] in $t,
          $pos1 < $pos2
```

Die Beschreibung der beiden Anfragesprachen XQL und XML-QL zeigt, dass die Arbeit mit ihnen recht gewöhungsbedürftig ist. Dazu kommt das technische Problem, dass die bislang verfügbaren Implementationen sehr langsam sind. Dies ist vor allem dadurch bedingt, dass das XML-Dokument vor der Anfrageverarbeitung vollständig in den Hauptspeicher geladen werden muss, um die verwendeten Referenzen verfolgen zu können. Denn prinzipiell könnte eine Referenz von einem Element ganz zu Beginn eines Dokuments auf ein Element am Ende verweisen.

XPath (1999)

Die als Pfadbeschreibungssprache spezifizierte Sprache *XPath* hat den Standardisierungsprozess des W3C erfolgreich durchlaufen (Clark und DeRose 1999) und wird beispielsweise zur Pfadbeschreibung in der Transformationssprache XSLT oder als Anfragesprache in Datenbankprodukten eingesetzt (s.u.). XPath-Ausdrücke ähneln der XQL-Sprache, sind aber wesentlich mächtiger. Bestehen bleiben die Probleme bei der Verfolgung der Dominanzreferenzen. Zwar steht mit der `id()`-Funktion ein angenehmerer und zugleich effizient unterstützter Referenzmechanismus zur Verfügung, doch ist auch hier nur die direkte Dominanz ausdrückbar:

```
id(//nt[@cat='NP']/edge/@idref)[@pos='NN']
```

XSLT (1999)

Die Transformationssprache *XSLT* ist eigentlich zur Konvertierung von XML-Dokumenten in andere Formate gedacht, kann aber auch zur Abfrage verwendet werden. Sie ist vollständig standardisiert (vgl. Clark 1999) und setzt XPath zur Beschreibung von Pfadausdrücken ein.

Das folgende Stylesheet zeigt, wie die Dominanzbeziehung rekursiv verfolgt werden kann. Es sucht nach einer Nominalphrase, die beliebig tief ein Nomen einbettet. Solange nur ein Nichtterminalknoten erreicht ist, wird die Suche rekursiv fortgesetzt. Sobald ein Terminalknoten erreicht wird, kann die Prüfung der Wortart erfolgen.

```
<xsl:template match="corpus">
  <xsl:apply-templates select="id(//nt[@cat='NP']/edge/@idref)"/>
</xsl:template>

<xsl:template match="nt">
  <xsl:apply-templates select="id(edge/@idref)"/>
</xsl:template>

<xsl:template match="t">
  <xsl:if test="@pos='NN'">
    Gefunden: <xsl:value-of select="@id" />
  </xsl:if>
</xsl:template>
```

Da die Transformationssprache XSLT erweiterbar ist, könnte eine umfangreiche Bibliothek von Templates und Funktionen zur Vereinfachung der Suchformulierung zusammengestellt werden. Das technische Problem besteht darin,

dass auch im Falle von XSLT stets der gesamte Dokumentbaum in den Hauptspeicher gelesen wird. Alle bislang zur Verfügung stehenden Implementationen sind daher recht langsam.

Datenbankbasierte Ansätze (2001)

Dem großen Interesse an XML-basierten Lösungen konnten sich auch die Datenbank-Hersteller nicht verschließen. Die Unterstützung von XML reicht dabei von der simplen Möglichkeit, ein XML-Dokument als Textblock in einer Tabellenzelle abzulegen, bis hin zur reinen XML-Datenbank. Die dabei zu findenen Hauptströmungen und ihre Verwendbarkeit in dieser Arbeit werden in diesem Abschnitt diskutiert.

Die ideale Form der XML-Unterstützung besteht in einer reinen XML-Datenbank. Kommerzielle Lösungen wie der *Tamino XML Server* der Software AG (`http://www.tamino.com`) bieten zum Datenzugriff neben Programmierschnittstellen auch XML-Anfragesprachen wie XPath. Eine Unterstützung für XML Query ist geplant. OpenSource-Lösungen wie *Xindice* (`http://www.apache.org/xindice`, zuvor unter dem Namen *dbXML* entwickelt) müssen mit einem schmaleren Spektrum an Funktionalität auskommen. Mit Hilfe von XPath-Anfragen und dem Datenzugriff über die standardisierte DOM-Schnittstelle (vgl. Apparao et al. 1998) lassen sich aber auch hier mächtige Anfrageinstrumente entwickeln (vgl. Eschweiler 2001b).

Als generelle Nachteile von XML-Datenbanken werden in der aktuellen Diskussion folgende Punkte genannt:

- Im Gegensatz zu SQL bei relationalen Datenbanken gibt es für XML-Datenbanken noch keine ausreichend mächtige und zugleich standardisierte Anfragesprache. Allerdings befindet sich mit *XMLQuery* eine Anfragesprache im Standardisierungsprozess, die diese Rolle in naher Zukunft einnehmen könnte (s.u.).

- Im Gegensatz zu relationalen Datenbanken, die auf einem einheitlichen mathematischen Modell arbeiten, ist die interne Arbeitsweise der XML-Datenbank-Implementationen nicht vorgegeben, sondern liegt vollständig in der Hand des Datenbank-Herstellers. Im Falle eines ineffizienten Zugriffs kann der Anwender weder Rückschlüsse auf mögliche Ursachen ziehen noch kann er bei der Formulierung von Anfragen die Eigenschaften des zugrunde liegenden Datenmodells zur Vermeidung solcher Problematiken berücksichtigen.

- Eine relationale Datenbank ist bei den meisten Anwendern bereits vorhanden. Bei der Verwendung einer reinen XML-Datenbank entsteht zusätzlicher Installations- und Administrationsaufwand.

Eine alternative Vorgehensweise besteht darin, eine vorhandene relationale Datenbank zur effizienten Repräsentation von XML-Dokumenten zu nutzen. Ein zentrales Problem bei der Verwaltung von XML-Dokumenten ist der Speicherbedarf zur Dokumentenrepräsentation. Der in Projekten wie *DBDOM* verfolgte Ansatz besteht darin, die Datenbank zur Verwaltung der Daten einzusetzen und dem Benutzer eine die Datenbank ummantelnde Daten-Schnittstelle zur Verfügung zu stellen (vgl. Eschweiler 2001a). In der Regel wird hier die DOM-Schnittstelle verwendet. Die Implementation der Schnittstelle erfolgt dabei so geschickt, dass Speicherbedarf und Zugriffszeit minimal gehalten werden können. Bei einer solchen Systemarchitektur stellt sich allerdings stets die Frage, ob im vorliegenden Kontext eines Baumbankanfrage-Werkzeugs mit einer speziell entwickelten Abbildung auf die relationale Datenbank wie im Falle von VIQTORYA (vgl. Abschnitt 3.5) nicht bessere Laufzeiten erreicht werden können.

Neben solchen Datenbankaufsätzen gibt es mittlerweile den Trend der Datenbankhersteller, XML direkt zu unterstützen. Die Lösungen waren aber bislang zu einfach realisiert, als dass sie an die Mächtigkeit von Speziallösungen heranreichen konnten. Mit Oracle hat der erste große Datenbankhersteller eine vollständige Integration von XML erreicht (vgl. Jansen 2002). Bis OpenSource-Lösungen wie mysql eine vergleichbare Unterstützung anbieten können, wird aber noch einige Zeit vergehen.

Es bleibt festzuhalten, dass Datenbanken zur Verwaltung einer XML-kodierten Baumbank eingesetzt werden können. Die Investitionssicherheit für eine reine XML-Datenbank ist zur Zeit noch zu niedrig und steht in einem schlechten Verhältnis zum zusätzlichen Aufwand. Datenbankaufsatz-Lösungen sind vielversprechend, sind aber letztlich im Kontext dieser Arbeit mit Speziallösungen wie VIQTORYA vergleichbar. Um alle Parameter der Implementation selbst in der Hand zu behalten, ist eine individuelle Lösung zu bevorzugen.

XML Query (2002)

Die Vielzahl der entwickelten XML-Anfragesprachen hat sich insgesamt eher kontraproduktiv ausgewirkt. Bedingt durch die Fülle sehr unterschiedlicher Ansätze haben sich viele Projekte gegen die Verwendung solcher proprietären Lösungen ausgesprochen. Diese Situation war Anlass für eine Standardisierungsinitiative des W3C.

Zwischenergebnis dieser Initiative war eine Spezifikation der Anforderungen, die an eine standardisierte XML-Anfragesprache gestellt werden (Chamberlin et al. 2001). Die seitdem entwickelte Anfragesprache *XML Query* soll einen vernünftigen Kompromiss zwischen allen bisherigen Strömungen darstellen (Boag et al. 2002).

Die aktuelle Spezifikation vom November 2002 ist bereits sehr stabil und lässt erste Rückschlüsse auf die Qualitäten von XML Query zu. Einige Experten

sprechen bereits davon, dass XML Query eine ähnliche Beachtung finden wird wie SQL für relationale Datenbanken. Es ist damit zu rechnen, dass der Standardisierungsprozess Anfang 2003 abgeschlossen ist und im weiteren Verlauf des Jahres erste vollständige Implementationenen erhältlich sein werden.

Zu den herausragenden Eigenschaften von XML Query gehören eine breite Palette von Datentypen und Funktionen, die Definition eigener Funktionen durch den Anwender und die Bereitstellung eines Allquantors. Für eine Einführung in XML Query an einem konkreten Anwendungsbeispiel sei auf Lezius (2003) verwiesen.

Um die Bedeutung von XML Query für die vorliegende Arbeit aufzuzeigen, ist im Folgenden eine Anfrage abgedruckt, die nach einer Nominalphrase sucht, die beliebig tief ein Adjektiv einbettet. Vorausgesetzt wird ein XML-Dokument `test.xml`, das das Korpus nach dem TIGER-XML-Format (vgl. Kapitel 7) kodiert. Die beiden Funktionen zum Testen einer Dominanzbeziehung zu Beginn zeigen, wie leicht der Sprachumfang durch selbstdefinierte Funktionen erweitert werden kann. Die eigentliche Anfrage durchläuft nun der Reihe nach alle Korpusgraphen. Für jeden Graphen werden zunächst alle Terminale und Nichtterminale bestimmt. Anschließend werden die beiden Variablen `$nt` und `$t` definiert, die die Nominalphrase bzw. das Adjektiv repräsentieren. Wesentlicher Bestandteil der Anfrage ist die Formulierung der Bedingung (Schlüsselwort `WHERE`), die die Attribute der beiden Knoten spezifiziert und eine allgemeine Dominanz zwischen den Knoten festlegt. Der abschließende Teil der Anfrage dient der Ausgabe der ermittelten Matches.

```
{-- Funktionsbibliothek --}

{-- Funktion zur Prüfung direkter Dominanz --}
define function dominatesImmediately(element $top,
                                     element $down) {
  some $edge in $top/edge
  satisfies ($edge/@idref = $down/@id)
}

{-- Funktion zur Prüfung allgemeiner Dominanz --}
define function dominatesGeneral(element $top,
                                 element $down,
                                 element* $nts) {
  if (dominatesImmediately($top, $down)) then (1)
  else ( some $node in $nts
         satisfies (dominatesImmediately($top,$node) and
                    dominatesGeneral($node,$down,$nts))
       )
}
```

```
{-- Anfrage --}

for $sentence in document("test.xml")/corpus/body/s

let $terminals := $sentence/graph/terminals/t
let $nonterminals := $sentence/graph/nonterminals/nt

for $nt in $nonterminals,
    $t in $terminals

where $nt/@cat="NP" and
      $t/@pos="ADJA" and
      dominatesGeneral($nt,$t,$nonterminals)

return <s id="{$sentence/@id}">
         <var name="nt">{$nt/@id}</var>
         <var name="t">{$t/@id}</var>
       </s>
```

Auf den ersten Blick mag die Beispielanfrage sehr komplex und unübersichtlich erscheinen. XML Query ist sicherlich auch als Anfragesprache für ein Baumbank-Suchwerkzeug wenig geeignet. Beeindruckend ist jedoch die Ausdruckskraft und die Modularität. Die Verwendung einer XML-Datenbank, die mit Hilfe von XML Query abgefragt werden kann, in Verbindung mit einer linguistisch motivierten Anfragesprache als Front-End (vergleichbar mit der Architektur von VIQTORYA) könnte deshalb in der Zukunft eine interessante Vorgehensweise darstellen.

Fazit

Alle bislang entwickelten Anfragesprachen liefern zwar brauchbare Ansätze, ihr Praxiseinsatz scheitert aber entweder an der mangelnden Unterstützung spezieller Bedürfnisse wie der IDREF-Dereferenzierung oder an Implementationen, die noch nicht ausgereift sind.

Mit XML Query scheint endlich eine Anfragesprache auf den Weg gebracht zu sein, die wirklich alle Anforderungen erfüllt. Da sie zudem von einer breiten Anwenderschaft getragen wird, ist mit der raschen Entwicklung effizienter Suchwerkzeuge zu rechnen. Für die vorliegende Arbeit könnten diese Entwicklungen in der Zukunft als alternative Vorgehensweise interessant werden.

3.8 Weitere Suchwerkzeuge

In diesem Abschnitt werden einige weitere Werkzeuge vorgestellt, die entweder weniger verbreitet sind oder in einem anderen Anwendungszusammenhang entstanden sind und damit nur begrenzt auf eine Baumbank angewendet werden können.

LDB

Das Suchwerkzeug *Linguistic DataBase* LDB ist eine datenbankbasierte Implementation (van Halteren 1997). Sie bildet eine Baumbank auf eine relationale Datenbank ab. Anfragen in einer logischen Anfragesprachen werden anschließend in SQL-Anfragen transformiert. Die Vorgehensweise ist damit mit der Vorgehensweise von VIQTORYA vergleichbar. Das im LDB-Projekt verwendete Datenmodell sieht geordnete Bäume vor, die auch ambig sein dürfen. Das verwendete Datenmodell ist für die hier benötigten Zwecke nicht allgemein genug.

VENONA

Das *VENONA*-System ist eine Implementation, die auf eine Baumbank für althebräische und ugaritische Texte zugeschnitten ist (vgl. Argenton 1998). Sie ist daher nur bedingt für eine andere Baumbank anpassbar. Interessant an dieser Arbeit sind die verwendeten Indexierungsstrategien. Argenton entwickelt in seiner Arbeit mit dem Baumgramm-Filter für Baumstrukturen eine zum n-Gramm-Textfilter analoge Filterstrategie, durch die eine erfolgreiche Suchraumreduktion erreicht wird. Die Arbeit von Argenton war eine wichtige Anregung für die Strategien zur Filterung des Suchraums, die in Kapitel 13 beschrieben werden.

TTS

Die *Treebank Tool Suite* stellt ein grafisches Interface für Baumbanken zur Verfügung, die nach dem Vorbild der PennTreebank annotiert worden sind (Cahill und van Genabith 2002). Gesucht wird in der Baumbank nach allen Instanzen einer kontextfreien Regel. Die im Korpus auftretenden Regeln werden dazu vorab ermittelt und dem Benutzer zur Auswahl präsentiert. Die Suche lässt zudem die Einschränkung zu, nur nach Instanzen zu suchen, die vorgegebene Wortformen umfassen. Da das System über einen Browser bedient wird, ist es prinzipiell auf jedem Rechner lauffähig.

Annotation graphs

Das allgemeine Datenmodell der *annotation graphs* ist bereits in Abschnitt 2.4 vorgestellt und diskutiert worden. Für dieses Datenmodell befindet sich ein Anfragewerkzeug in Vorbereitung, das auch für die Suche auf Baumbanken interessant sein könnte. Eine Spezifikation der benötigten Ausdruckskraft und Skizze der geplanten Implementation ist in Bird et al. (2000) zu finden.

MATE

Die *MATE-Workbench* ist ein Werkzeug zur Annotation von gesprochener und geschriebener Sprache (McKelvie et al. 2001). Es bietet Funktionen zur Darstellung und Bearbeitung von Annotationen und zusätzlich zur Auswertung komplexer Suchanfragen auf vorhandenen Korpora. Das dazu implementierte Anfragewerkzeug arbeitet auf einem XML-Dokument, das die linguistische Annotation trägt (Heid und Mengel 1999). Auch Graphstrukturen können mit diesem Werkzeug verarbeitet werden. Interessant ist auch eine grafische Eingabehilfe für Suchanfragen, die bzgl. des Konzepts der Oberfläche von VIQTORYA ähnelt.

3.9 Zusammenfassung

Das vorliegende Kapitel hat einige interessante Arbeiten zur Suche auf syntaktisch annotierten Korpora beschrieben. Die Verwendung in der vorliegenden Arbeit scheitert in den meisten Fällen am zu engen Datenmodell. Allein das VIQTORYA-System unterstützt das erforderliche Datenmodell in weiten Teilen, ist aber für die Verwendung beliebiger Korpora nicht flexibel genug. Aus diesen Gründen wird in der vorliegenden Arbeit neben einer eigenen Anfragesprache auch eine Anfrageverarbeitungs-Komponente entwickelt.

Aus der Diskussion der verschiedenen Werkzeuge sind einige Erkenntnisse hervorgegangen, die die weitere Arbeit maßgeblich beeinflussen werden:

- Die grafische Benutzeroberfläche entscheidet mit über die Verwendbarkeit eines Werkzeugs. Sie ist zwar keine notwendige Voraussetzung, macht das Arbeiten aber in jedem Falle angenehmer.
- Die grafische Eingabe von Benutzeranfragen stellt für Gelegenheitsanwender eine interessante Herangehensweise an ein Anfragewerkzeug dar.
- Die XML-Anfragesprache XML Query wird in naher Zukunft einen entscheidenden Standard darstellen. Die Entwicklungen in diesem Bereich sind sehr interessant und sollten weiter verfolgt werden.

Teil II

Eine Korpusbeschreibungssprache

Kapitel 4

Vorüberlegungen

Nachdem mit den gängigen Korpusformaten und Anfragewerkzeugen relevante Vorarbeiten vorgestellt wurden, geht es in diesem Teil der Arbeit um die äußere Konzeption des zu entwickelnden Suchwerkzeugs. Aspekte der Benutzeroberfläche werden in Teil IV behandelt.

In diesem Teil der Arbeit werden zwei Entscheidungen getroffen:

1. Format zur Korpusdefinition

   In welchem Format werden Korpora definiert, die mittels des Anfragewerkzeugs abgefragt werden können? Dieses Format muss die Kodierung von Baumbanken erlauben, die sich nach dem gewählten Datenmodell richten (vgl. Abschnitt 1.3).

2. Format zur Korpusanfrage

   Wie sieht die Anfragesprache aus, die zur Abfrage der Korpora dienen soll? Auch hier muss die Anfragesprache die Eigenheiten des Datenmodells berücksichtigen, also beispielsweise die Abfrage von Kantenbeschriftungen erlauben.

Das vorliegende Kapitel zeigt ein Konzept auf, das die beiden zentralen Fragen mit der Idee einer gemeinsamen Korpusbeschreibungssprache beantwortet. Die beiden folgenden Kapitel 5 und 6 beschreiben die resultierende TIGER-Korpusbeschreibungssprache informell und formal. Kapitel 7 definiert mit TIGER-XML eine zur TIGER-Korpusbeschreibungssprache äquivalente XML-basierte Sprache zur Definition von Korpora. Die technischen Gründe, die zum Übergang zu einem XML-basierten Format geführt haben, werden dort ausführlich dargelegt. Das abschließende Kapitel 8 zeigt, in welchen Teilen der Implementation sich die vorher entwickelten Ideen wiederfinden.

Das vorliegende Kapitel legt zunächst das Konzept der Korpusbeschreibungssprache fest. Im ersten Abschnitt dieses Kapitels wird das Spannungsfeld Korpusdefinition vs. Korpusabfrage genauer beleuchtet. Abschnitt 4.2 schlägt

dann eine gemeinsame Beschreibungssprache als Konzept vor. Der nachfolgende Abschnitt 4.3 entwickelt Anforderungen an eine Beschreibungssprache, die bei der Konzeption berücksichtigt werden sollen.

4.1 Korpusdefinition vs. Korpusabfrage

Die Problemfelder Korpusdefinition und Korpusabfrage werden in den vorgestellten Suchwerkzeugen stets unabhängig voneinander betrachtet. Wie in Kapitel 3 beschrieben wird, dient etwa für die Werkzeuge *CorpusSearch* und *tgrep* das Klammerstrukturformat als Korpuseingangsformat; die Anfragesprachen hingegen wurden separat entwickelt und sind von diesem Format verschieden.

```
( (S (NP (NP (DT A)
             (JJ new)
             (NN project))
         (VP (VBN planned)
             (NP (-NONE- *))))
     (VP (VBZ is)
         (NP (NP (DT the)
                     (NN use))
                 (PP (IN of)
                     (NP (NNP Bio-Dynamic)
                         (NNP Starter)))))
     (. .)) )
```

Abbildung 4.1: Ein Beispielsatz der PennTreebank (Brown-Korpus)

In Abb. 4.1 ist ein Beispielsatz der PennTreebank im Klammerstrukturformat dargestellt. Die Anfragen nach einer Nominalphrase, die beliebig tief eine Nominalphrase und eine Verbalphrase einbettet, lauten für die beiden Werkzeuge *CorpusSearch* und *tgrep* folgendermaßen (Beispielsatz in Abb. 4.1 enthält die gesuchte Struktur):

CorpusSearch: `([1]NP dominates NP) AND ([1]NP dominates VP)`
tgrep: `NP << NP << VP`

Wie dieses Beispiel zeigt, können Korpusformat und Anfragesprache weit auseinander liegen. Dabei hat die strikte Trennung technische und konzeptionelle Nachteile:

- Für das Korpusformat sowie die Anfragesprache müssen zwei voneinander unabhängige Parser implementiert werden.

- Der Benutzer muss sich mit zwei verschiedenen Formaten auseinandersetzen.
- Der formale Nachweis der Korrektheit und Vollständigkeit des Suchwerkzeugs ist aufwändiger, da die formale Semantik zweier Repräsentationsformalismen angegeben werden muss.

4.2 Die Idee einer Korpusbeschreibungssprache

Für das in dieser Arbeit beschriebene Anfragewerkzeug wird versucht, die Trennung von Korpusdefinitionssprache und Korpusanfragesprache aufzuheben. Die so entstandene Sprache wird als **Korpusbeschreibungssprache** bezeichnet. Dieser Begriff soll betonen, dass in der gemeinsamen Sprache auf der einen Seite das Korpus durch eine vollständige Spezifikation der Korpusgraphen definiert wird, auf der anderen Seite bei der Suche unterspezifizierte Graphenfragmente beschrieben werden.

Die oben aufgeführten Nachteile werden mit dieser Konzeption vermieden. Zudem ist das Design der beiden Formate als Einheit ist in sich geschlossener, da die Korpusdefinitionssprache eine echte Teilmenge der Korpusanfragesprache darstellt. Der Unterschied besteht lediglich im Grad der Spezifikation (vollständig spezifiziert vs. unterspezifiziert).

4.3 Anforderungen an eine Korpusbeschreibungssprache

Der Entwurf einer formalen Sprache ist eine Aufgabe, die oft unterschätzt wird. Eine für den Entwickler intuitive Sprache muss nicht automatisch auch auf die späteren Benutzer intuitiv wirken. Dazu kommt das Problem, dass die Sprache nach einer erster Veröffentlichung möglichst nicht geändert werden sollte, da ansonsten Umgewöhnungsprobleme der Benutzer die Akzeptanz des gesamten Werkzeugs gefährden.

Für den Entwurf der Korpusbeschreibungssprache gilt es deshalb zunächst, Arbeiten zum Design von Anfragesprachen zu Rate zu ziehen, um Anforderungen an die Sprache festzulegen. Arbeiten in diesem Umfeld stammen vor allem aus den etablierten Bereichen der Datenbanken (Vossen 1994, Kapitel 15) und des Information Retrieval (Baeza-Yates und Ribeiro-Neto 1999, Kapitel 4). Daneben hat das große kommerzielle Interesse an XML zu Sprachentwürfen und Kriterienkatalogen für XML-Anfragesprachen geführt (vgl. W3C 1998, Chamberlin et al. 2001).

Da die meisten entwickelten Kriterien nicht vom verwendeten Datenmodell abhängen, sind diese Arbeiten auch auf das Korpusbeschreibungsproblem an-

wendbar. Dies hat zu folgendem Anforderungskatalog für eine Korpusbeschreibungssprache geführt:

1. Ausdruckskraft

   Die Mindestanforderungen an die Ausdruckskraft der Anfragesprache werden durch die Korpusdefinition bestimmt. Für die Korpusdefinition bedeutet minimale Ausdruckskraft, dass die Einheiten der Baumbank in Bezug auf ihr Datenmodell vollständig beschrieben werden können: Können Knoten und ihre Attribute definiert werden? Lassen sich Beziehungen zwischen den Knoten spezifizieren? Sind Kantenbeschriftungen möglich?

   Von der Ausdrucksstärke der Anfragesprache hängt unmittelbar der praktische Nutzen des gesamten Anfragewerkzeugs ab: Je umfangreicher die Möglichkeiten zur Suche nach Strukturen, desto höher der praktische Nutzen. Beispiele für Ausdrucksmittel sind Möglichkeiten der Unterspezifikation (disjunktive Verknüpfung, reguläre Ausdrücke usw.) oder die Einführung von Quantoren.

   Dem verständlichen Wunsch der Benutzer nach Mächtigkeit der Anfragesprache steht aber die Sicherstellung einer effizienten maschinellen Anfrageverarbeitung entgegen. Ein Allquantor beispielsweise ist ein ausdrucksstarkes Hilfsmittel, bereitet aber Schwierigkeiten für die effiziente Verarbeitung. Da (partielle) Ineffizienz die Akzeptanz des gesamten Werkzeugs in Frage stellt, muss beim Entwurf der Sprache stets sorgfältig abgewogen werden, ob Ausdruckskraft oder Effizienz im Vordergrund stehen soll.

2. Übersichtlichkeit

   Für die Verarbeitung einer Korpusdefinition oder einer Korpusanfrage durch ein Programm ist eine eindimensionale lineare Darstellung nötig. Syntaxgraphen sind aber per Definition zweidimensional (vgl. Abschnitt 1.2). Die Abbildung der zweidimensionalen Graphen auf die eindimensionale Textebene stellt daher ein zentrales Problem dar.

   Im folgenden Beispiel sind zwar die Elementarsymbole durch den Betrachter leicht zu isolieren, das Ablesen der Graphstrukturen ist hingegen deutlich schwieriger (Beispiel entnommen aus Calder 1999):

   [S [NP [PN Etta]] [VP [V1 chased] [NP [Det a][N bird]]]]

   Eine Möglichkeit, zu übersichtlicheren Korpusbeschreibungen zu kommen, besteht in der Verwendung von Modularisierungsmechanismen. Hier werden umfangreiche, komplexe Anfragebestandteile durch abkürzende Schreibweisen ersetzt.

3. Transparenz

   Eine wesentliche Konsequenz aus der genannten Problematik besteht darin, die formale Syntax und Semantik der Korpusbeschreibungssprache explizit festzulegen. Durch diese Festlegung werden Missverständnisse vermieden und damit Transparenz gegenüber dem Anwender geschaffen.

4. Erlernbarkeit

   Übersichtlichkeit und Transparenz sind noch keine Garantie für die leichte Erlernbarkeit einer Sprache. Es gilt hier, auf die Vorkenntnisse der Zielgruppe einzugehen: Welche Formalismen und Anfragesprachen sind in der Zielgruppe verbreitet? Ist eine Orientierung an diesen Darstellungsformen sinnvoll?

   Da Baumbanken syntaktische Informationen repräsentieren, ist eine Orientierung an Grammatikformalismen hilfreich. Als verbreitete Anfragewerkzeuge können die in Kapitel 3 vorgestellten Werkzeuge genannt werden.

5. Verarbeitbarkeit

   Neben der menschlichen Verarbeitbarkeit spielt auch die maschinelle Verarbeitbarkeit eine wichtige Rolle. Ausdrücke solltes leicht parsebar sein, die Kodierung von Sonderzeichen (insbesondere beliebiger Unicode-Zeichen) problemlos möglich sein.

Kapitel 5

Konzeption der Korpusbeschreibungssprache

Das Ziel dieses Kapitels besteht darin, den Konzeptionsprozess der Korpusbeschreibungssprache nachzuzeichnen. Es geht damit nicht um eine Einführung in die Beschreibungssprache – eine didaktisch aufbereitete Einführung für die Benutzer der TIGERSearch-Software ist in König und Lezius (2002a) zu finden. In diesem Kapitel werden vielmehr die wesentlichen Sprachelemente vorgestellt und die Überlegungen und Motivationen beschrieben, die zur Festlegung dieser Sprachelemente geführt haben.

Die Konzeption der Sprache ist maßgeblich durch Arbeiten auf den Gebieten der Baumbeschreibungssprachen (Rogers und Vijay-Shanker 1992, Blackburn et al. 1993, Duchier und Niehren 1999) und der unifikationsbasierten Formalismen (Höhfeld und Smolka 1988, Dörre und Dorna 1993, Dörre 1996, Dörre et al. 1996, Emele 1997) sowie Arbeiten im Bereich Korpusanfragesprachen (Christ 1994, Christ et al. 1999) beeinflusst worden.

Dieses Kapitel geht auf einen Artikel zurück, der die Hauptideen des Sprachentwurfs beschreibt (König und Lezius 2000). Die Beispielgraphen, die zur Illustration verwendet werden, sind nach den TIGER-Annotationsrichtlinien (Albert et al. 2002) annotiert worden. Alle Wortarten-Tags, die in diesen Beispielgraphen sowie in Anfragebeispielen verwendet werden, richten sich nach dem STTS-Tagset (Thielen et al. 1999).

5.1 Knoten, Relationen, Graphen

Eine Baumbank besteht aus einer Sammlung von Graphstrukturen. Es ist daher naheliegend, beim Entwurf der Beschreibungssprache eine graphentheoretische Sichtweise zu verfolgen. Die Graphentheorie hat zu allgemein akzeptierten Begrifflichkeiten und Sichtweisen geführt. Es ist daher sinnvoll, das Rad nicht neu zu erfinden und sich stattdessen an diesen Konventionen zu orientieren.

Grundbestandteile der Sprache sind damit Knoten (vgl. 5.1.1), die im vorliegenden Datenmodell auf der Wortebene und im Inneren des Graphen verwendet werden (Terminal- und Nichtterminal-Knoten). Knoten werden über Relationen miteinander verknüpft (vgl. 5.1.2). Eine Aufzählung von Knoten und Relationen beschreibt dann einen (Teil-)Graphen (vgl. 5.1.3).

5.1.1 Knoten

Viele Anfragesprachen erlauben eine schrittweise Spezifikation eines Knotens, ohne dass die Teilspezifikationen in unmittelbarer Nähe stehen müssen. Die Anfragesprache des VIQTORIA-Systems beispielsweise lässt den folgenden Ausdruck zu (vgl. Abschnitt 3.5):

```
a.word="Haus" AND b.pos="NE" AND c.pos="NN" AND a.pos="NN"
```

Die Semantik eines solchen Ausdrucks ist schwer erkennbar, da die zusammengehörigen Einheiten weit auseinanderliegen (hier: Knoten `a`). Für die Beschreibungsprache ist es daher sinnvoll, dass alle Spezifikationen eines Knotens in einer Texteinheit erfolgen.

Die Knoten des Datenmodells, die Phrasen bzw. Token darstellen, werden bei der Annotation mit morpho-syntaktischen Attributen wie Wortart, Kasus usw. versehen. Für die Kodierung solcher Informationen werden in der Computerlinguistik Merkmalsstrukturen verwendet (auch als Featurestrukturen bezeichnet; für eine Einführung siehe Kolb 2001). Bei den Knoten des vorliegenden Datenmodells handelt es sich dabei um (flache) Merkmalsstrukturen, in denen Attributwerte lediglich Konstanten oder Typen (vgl. Abschnitt 5.3) sein dürfen. Die Art der Knotenbeschreibung durch Attribut-Wert-Paare, eingegrenzt durch eckige Klammern, orientiert sich deshalb an der Schreibweise für Merkmalsstrukturen. Das folgende Beispiel zeigt eine Knotenbeschreibung als Ausdruck der Beschreibungssprache und als Merkmalsstruktur:

```
[word="Haus" & pos="NN"]
```
(5.1)

$$\begin{bmatrix} \text{WORD} & \text{Haus} \\ \text{POS} & \text{NN} \end{bmatrix} \tag{5.2}$$

Die einzelnen Bestandteile einer Knotenbeschreibung werden im Folgenden näher erläutert.

Attribut-Wert-Paare

Im Innern einer Knotenbeschreibung sind Attribut-Wert-Paare angegeben, die durch boolesche Ausdrücke zu einer Attribut-Spezifikation verknüpft werden.

Das Gleichheitszeichen muss explizit angegeben werden, da auch Ungleichheit ausgedrückt werden kann:

```
pos = "NN"
```
(5.3)

```
pos != "NN"
```
(5.4)

Konstanten werden explizit durch Anführungsstriche markiert, um sie von Typsymbolen unterscheiden zu können (vgl. Abschnitt 5.3). Diese Schreibweise ist auch in anderen Anfragewerkzeugen wie CQP verbreitet (vgl. Abschnitt 3.1).

Damit die Zuordnung der Spezifikationen zu den einzelnen Attributen übersichtlich bleibt, sind bereits auf Attributwert-Ebene boolesche Ausdrücke zugelassen. Statt der getrennten Schreibweise

```
(pos = "NN") | (pos = "NE")
```
(5.5)

kann hier kompakter formuliert werden:

```
pos = ("NN"|"NE")
```
(5.6)

Auch negierte Konstanten oder Typen können verwendet werden:

```
pos = (!noun & !"ADJA")
```
(5.7)

Neben Gleichheit und Ungleichheit sind für ein Attribut-Wert-Paar auch Match bzw. Nicht-Match gegen einen regulären Ausdruck zugelassen. Die Schreibweise `/expr/` für reguläre Ausdrücke orientiert sich dabei an der Programmiersprache Perl, die regulären Ausdrücke selbst richten sich nach den Perl5-Konventionen (vgl. Wall et al. 2000):

```
word = /Haus.*/
```
(5.8)

Die Erfahrungen mit dem Korpusanfragetool CQP haben gezeigt, dass viele Benutzer mit den vielfältigen Ausdrucksmitteln regulärer Ausdrücke nicht vertraut sind. Insbesondere wird oft vermutet, dass durch `word=/Haus/` ein exakter Match ausgedrückt wird. Tatsächlich bedeutet dieser Ausdruck nur, dass `Haus` in einer Wortform enthalten ist. Um den Anwendern entgegenzukommen, wird der Ausdruck `/Haus/` intern als `/^Haus$/` verarbeitet, d.h. der Attributwert ist gleich `Haus`.

Attribut-Spezifikation

Eine Attribut-Spezifikation besteht aus booleschen Ausdrücken über Attribut-Wert-Paaren. Diese Form der Verknüpfung findet sich in den meisten Anfragesprachen, z.B. im VIQTORIA-System (vgl. Anfragebeispiel in Abschnitt 3.5).

```
[word="Haus" & pos="NE"]
```
(5.9)

```
[!(word="Haus" & pos="NN") | (word="Mann")]
```
(5.10)

Die maximal unterspezifizierte Attribut-Spezifikation wird als `[]` geschrieben.

Knotenbeschreibung

Jeder Knoten eines Korpus wird in eindeutiger Weise durch einen Identifier (ID) beschrieben. Die Auflistung der Attribut-Wert-Paare reicht zur eindeutigen Unterscheidung zweier Knoten nicht aus, da sie dieselben Attributwerte besitzen können. Eine Knotenbeschreibung besteht damit aus der Knoten-ID zusammen mit der Attribut-Spezifikation:

```
"t1":[word="Haus" & pos="NN"]
```
(5.11)

Auf den ersten Blick orientiert sich das Konzept der Knotenbeschreibung nicht mehr an Merkmalsstrukturen. Doch die folgende Merkmalsstruktur repräsentiert exakt den obigen Knoten. Eine Knotenbeschreibung wird durch ihre ID und ihre Attributspezifikation bestimmt. Die Attributspezifikation besteht wiederum aus Attribut-Wert-Paaren:

$$\begin{bmatrix} \text{ID} & \text{t1} \\ \text{ATTR} & \begin{bmatrix} \text{WORD} & \text{Haus} \\ \text{POS} & \text{NN} \end{bmatrix} \end{bmatrix} \tag{5.12}$$

Während feste IDs nur bei der Korpusdefinition Sinn machen, können in Korpusanfragen auch Knotenvariablen verwendet werden (vgl. auch Abschnitt 5.4). Sie matchen (sofern ungebunden) eine beliebige Knoten-ID im Korpus.

```
#var:[word="Haus" & pos="NN"]
```
(5.13)

Fehlt die Knoten-ID bzw. die Knoten-Variable, so hat die Knotenbeschreibung eine unspezifizierte Knoten-ID und matcht jede Knoten-ID im Korpus.

```
[word="Haus" & pos="NN"]
```
(5.14)

Fehlt hingegen die Attribut-Spezifikation, so bleiben die Attribut-Werte unspezifiziert:

```
#var
```
(5.15)

5.1.2 Relationen

Das verwendete Datenmodell besteht aus Graphstrukturen, die zweidimensionale Objekte bilden (vgl. Abb. 5.1). Zur Beschreibung der horizontalen und vertikalen Graphebene werden in der Computerlinguistik in der Regel die Basisrelationen direkte Dominanz und lineare Präzedenz verwendet (vgl. Blackburn et al. 1993). Es ist daher sinnvoll, sich an diesen Begrifflichkeiten zu orientieren.

In den folgenden Abschnitten werden die beiden Basisrelationen bezüglich des Datenmodells definiert und davon abgeleitete Relationen präsentiert.

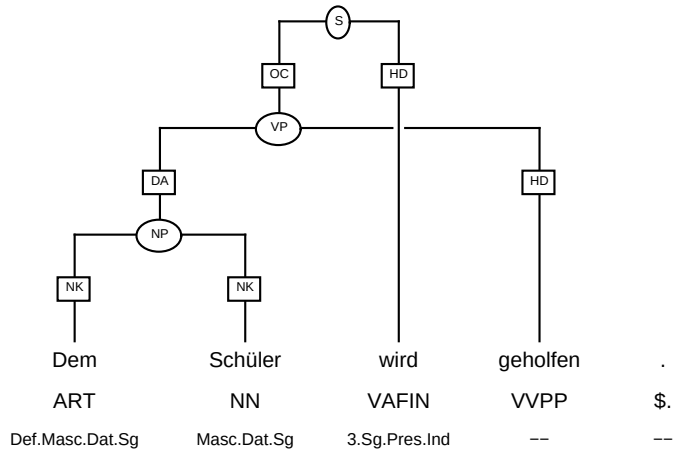


Abbildung 5.1: Ein Beispielsatz nach dem Datenmodell

Dominanz

Da das Datenmodell Kantenbeschriftungen zulässt (vgl. Abb. 5.1), wird als Basisrelation nicht die direkte Dominanz, sondern die beschriftete direkte Dominanz `>L` gewählt. Der Ausdruck

```
#v >HD #w
```
(5.16)

besagt, dass der Knoten `#v` über eine mit `HD` (Abkürzung für *head*) beschriftete Kante zum Knoten `#w` führt. Im Beispielsatz wird so eine Kopfbeziehung ausgedrückt. Die direkte Dominanz > ist damit eine abgeleitete Relation, die die Kantenbeschriftung unterspezifiziert lässt.

Es sei an dieser Stelle angemerkt, dass die Verwendung von Kantenbeschriftungen eine linguistische Entscheidung des Anwenders darstellt. Falls beispielsweise eine Modellierung syntaktischer Funktionen als Attribut sinnvoller erscheint, so steht dem Anwender diese Möglichkeit natürlich offen. Dabei ist allerdings zu bedenken, dass die Verwendung der beschrifteten Dominanzrelation intuitiver ist. Sie entspricht zudem der Darstellung sekundärer Kanten, für die eine alternative Modellierung als Attribut nicht möglich ist (s.u.).

Durch die Wahl des Relationssymbols > für die Dominanz soll ausgedrückt werden, dass der linke Knoten dem rechten übergeordnet ist. In *tgrep* (vgl. Abschnitt 3.2) wird für die Dominanz das Zeichen < mit einer ähnlichen Intuition verwendet (der linke Knoten umfasst den rechten Knoten). Um beiden Intuitionen gerecht zu werden, könnten beide Symbole synonym verwendet werden. Eine solche Lösung ist aber unglücklich, da zwei Benutzer sich schwer verständigen können, wenn sie unterschiedliche Schreibweisen verwenden. Eine pfadorientierte Schreibweise `#v / #w`, wie XML-Anfragesprachen sie zum Teil verwenden (vgl. Abschnitt 3.7), ist in der Computerlinguistik hingegen unüblich.

Die Wahl des Symbols für allgemeine Dominanz `>*` ist streng genommen inkonsequent, da die Basisrelation `>L` nicht reflexiv ist. Das Symbol für allgemeine Dominanz müsste damit in Anlehnung an reguläre Ausdrücke eigentlich `>+` heissen. Da aber wenige Anwender intensiv mit regulären Ausdrücken gearbeitet haben, der Stern als Zeichen für Beliebigkeit hingegen sehr verbreitet ist (z.B. bei Internet-Suchmaschinen), ist die Wahl zugunsten der Verständlichkeit und der leichteren Lesbarkeit auf das Symbol `>*` gefallen. Es sei noch erwähnt, dass die Schreibweisen für Dominanz in Stufen `>m` und `>m,n` an reguläre Ausdrücke angelehnt sind.

Die Beschreibung von Kantenbeschriftungen ist auf die Angabe einer Konstante beschränkt. Eine Disjunktion von Kantenbeschriftungen lässt sich jedoch auch über eine Disjunktion von Relationen unter der Verwendung von Knotenvariablen (vgl. Abschnitt 5.4) ausdrücken:

```
(#np:[cat="NP"] >HD [pos="NN"]) | (#np >SB [pos="NN"])
```

Mit der linken und rechten Ecke (`>@l` und `>@r`) stehen zwei weitere Dominanzrelationen zur Verfügung. Sie werden zum einen zur Definition der linearen Präzedenz gebraucht (s.u.). Sie erlauben zum anderen, die Projektion eines Knotens auf die Terminalebene zu beschreiben.

Präzedenz

Die Definition der linearen Präzedenz wird maßgeblich dadurch beeinflusst, dass das Datenmodell kreuzende Kanten erlaubt. Die beiden Visualisierungen in Abb. 5.2 machen deutlich, dass im Datenmodell zwar die Terminalknoten angeordnet sind, die Anordnung der Nichtterminale hingegen ein Graph**layout**problem darstellt. Denn beide Abbildungen zeigen eine korrekte Visualisierung desselben Graphen. Es stellt sich damit die Frage, wie die Präzedenz sinnvoll definiert werden kann.

Eine erste Idee zur Definition der Präzedenz zweier Knoten besteht in der Rückführung auf die Präzedenz der Projektionen (hier: Projektion auf die Terminalebene). Doch durch kreuzende Kanten kann es hier zu wenig intuitiven Aussagen kommen. So würde im abgebildeten Beispiel keine Präzedenzbeziehung zwischen den Knoten `#n2` und `#n3` gelten.

Um dieses Problem zu vermeiden, wird die Präzedenz zweier Knoten auf die Präzedenz ihrer linkesten Nachfahren (d.h. der linken Ecken) zurückgeführt. Die lineare Präzedenz `.` ist damit folgendermaßen definiert: Für zwei Knoten gilt `#v.#w` genau dann, wenn für den linkesten Nachfahren `#vl` von `#v` und den linkesten Nachfahren `#wl` von `#w` gilt: `#vl.#wl`. Die lineare Präzedenz der Terminalknoten `#vl` und `#wl` wird bei der Korpusdefinition festgelegt (vgl. Abschnitt 5.2). Die allgemeine Präzedenz `.*` gilt entsprechend, falls zwischen den linken Ecken eine allgemeine Präzedenzbeziehung besteht.

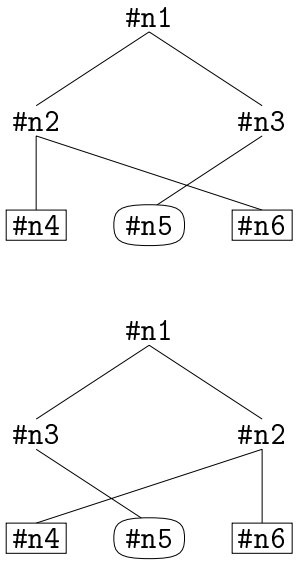


Abbildung 5.2: Visualisierungen eines Graphen

Ein Diskussionspunkt bei dieser Definition besteht darin, ob zwei Knoten, die in einer Dominanzbeziehung stehen, aus der Definition ausgenommen werden sollen. Im Beispiel würde etwa `#n1.*#n3` gelten, aber nicht `#n1.*#n2`. Allerdings kann der Anwender bei Bedarf die Forderung nach einem Ausschluss einer Dominanzbeziehung explizit formulieren: `#v.*#w & #v!>*#w` (vgl. Negation von Relationen in diesem Abschnitt). Um den Anwendern beide Möglichkeiten offen zu halten, ist die allgemeinere Definition festgelegt worden, die bei Bedarf über die zusätzliche Dominanzrelation eingeschränkt werden kann.

Es sei angemerkt, dass auch die erstgenannte Definition (Präzedenz der Projektionen) mit Hilfe linker und rechter Ecken ausgedrückt werden kann: Ein Knoten `#v` steht demnach vor einem Knoten `#w`, wenn der rechteste Nachfolger von `#v` vor dem linkesten Nachfolger von `#w` steht. Dem Anwender stehen somit alle Möglichkeiten zur Definition eines eigenen Präzedenzbegriffs zur Verfügung. Eine solche individuelle Definition kann auf elegante Weise in einem Template gekapselt werden (vgl. Abschnitt 5.5).

Das Relationssymbol `.` ist in vielen Programmiersprachen wie Perl ein gebräuchliches Symbol für Konkatenation. Es soll als Präzedenzsymbol andeuten, dass zwei Knoten nebeneinander stehen. Diese Schreibweise wird auch von *tgrep* verwendet. Eine weitere verbreitete Schreibweise für die Präzedenz ist das Zeichen `<` (vgl. Blackburn et al. 1993), das aber zu Verwechslungen mit dem Dominanzsymbol führt. Überlegungen, das Dominanzsymbol zugunsten dieser Schreibweise z.B. in `^` zu ändern, verlagern das Problem nur auf eine andere Ebene. Dazu kommt, dass das Zeichen `^` auf vielen Tastaturen nicht direkt

eingegeben werden kann. Damit die beiden Hauptsymbole der Sprache deutlich unterscheidbar und leicht über die Tastatur einzugeben sind, wurden > und . als Symbole festgelegt.

Während die Bedeutung der unmittelbaren linearen Präzedenz für zwei Wortknoten völlig intuitiv ist, gilt es zu beachten, dass unmittelbare lineare Präzedenz zweier Nichtterminale nur selten gilt, da die linkesten Nachfahren nur selten direkte Nachbarn sind. Stattdessen wird man für Nichtterminale meist auf die allgemeine Präzedenz `.*` zurückgreifen.

Geschwisterbeziehung

Die Geschwisterbeziehung lässt sich streng genommen durch direkte Dominanzbeziehungen ausdrücken. Zwei Knoten sind demnach Geschwister, wenn sie einen gemeinsamen Mutterknoten haben. Da die Geschwisterbeziehung häufig verwendet wird (z.B. zur Abfrage der Argumentstruktur in flach annotierten Korpora) und ein eigenes Symbol zur Übersichtlichkeit beiträgt, ist mit `$` eine eigene Geschwisterrelation eingeführt worden.

Die abgeleitete Relation `#v $.* #w` stellt eine Mischung zwischen Dominanz- und Präzedenzbeziehungen dar. Sie drückt aus, dass der Geschwisterknoten `#v` vor `#w` steht.

Eine Relation `$.` ist nicht definiert worden. Denn hier wird als Semantik leicht vermutet, dass die beiden Geschwisterknoten direkt benachbart sind, d.h. kein weiterer Geschwisterknoten zwischen ihnen steht. Und eben diese Semantik würde hier nicht gelten, da die Präzedenz der linkesten Nachfahren betrachtet wird. Um Verwirrungen zu vermeiden, ist auf diese Relation ganz verzichtet worden.

Sekundäre Kanten

Eine Besonderheit des Datenmodells ist die Integration sekundärer Kanten. Um diese deklarierbar bzw. abfragbar zu machen, ist eine weitere Relation vorgesehen, die sich direkt an die Definition der Dominanz anlehnt. Der Ausdruck

```
#v >~OA #w
```
(5.17)

besagt, dass der Knoten `#v` über eine mit `OA` beschriftete sekundäre Kante zum Knoten `#w` führt. Im Beispielsatz in Abb. 5.3 wird durch eine sekundäre Kante ausgedrückt, dass die koordinierte Nominalphrase *Äpfel und Birnen* Akkusativobjekt der Konstituente *Er kauft* ist. Die Relation `>~` ist wieder eine abgeleitete Relation, die die Kantenbeschriftung unterspezifiziert lässt.

Es sei an dieser Stelle angemerkt, dass sekundäre Kanten nicht über Attribute modelliert werden können, da ein Knoten sowohl beliebig viele ausgehende

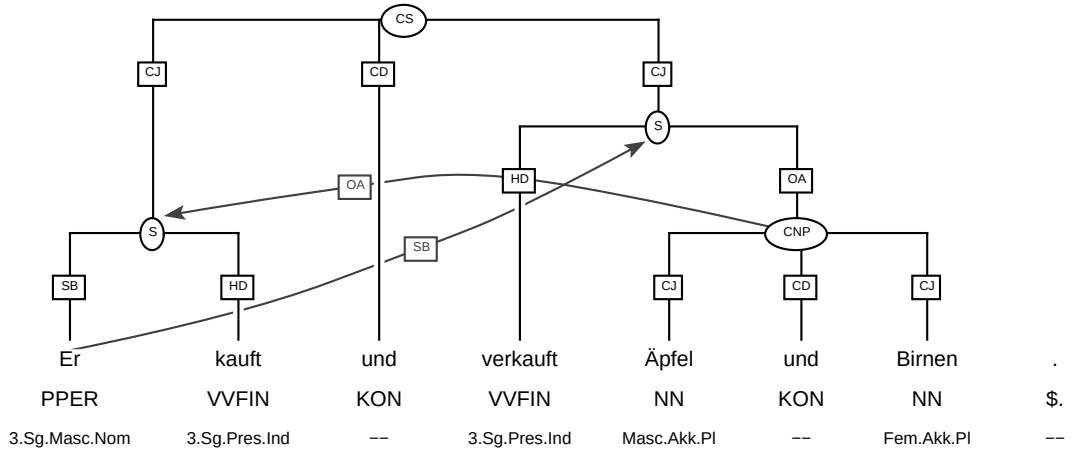


Abbildung 5.3: Ein Beispielsatz mit sekundären Kanten

als auch beliebig viele eingehende sekundäre Kanten umfassen kann. Die eingeführte Relation stellt damit ein notwendiges Sprachmittel zur Unterstützung sekundärer Kanten dar.

Negation der Relationen

Alle Relationen können auch negiert werden. Die Negation wird dabei stets am Relationssymbol festgemacht, eine Negation der gesamten Relation ist nicht möglich. Man betrachte als Motivation für diese Entscheidung den folgenden Ausdruck, der in der Beschreibungssprache nicht zulässig ist:

```
! ( [cat="NP"] > [pos="NN"] )
```

Die Semantik der Negation ist hier unklar: Sollen hier ein NP-Knoten und ein NN-Knoten existieren, die aber nicht in einer direkten Dominanzbeziehung stehen? Oder soll ein Nicht-NP-Knoten einen NN-Knoten dominieren bzw. ein NP-Knoten einen Nicht-NN-Knoten? Oder meint dieser Ausdruck gar, dass es einen NP-Knoten geben muss, dessen Töchter allesamt keine NN-Knoten sind? Die Beschreibungssprache kann zwar hier eine Festlegung treffen, doch kann eine solche Entscheidung kaum intuitiv sein.

Bedingt durch diese Problematik ist es sinnvoller, die Relationsnegation direkt am Relationssymbol festzumachen, den oft als wünschenswert angesehenen Allquantor jedoch ggf. separat einzuführen. Der folgende Ausdruck zeigt, dass die beiden Knoten existieren, aber nicht in einem Dominanz-Verhältnis stehen.

```
[cat="NP"] !> [pos="NN"]
```
(5.18)

Der Vollständigkeit halber sei noch darauf hingewiesen, dass aus analogen Gründen die Relation `$.*` als einzige nicht negiert werden kann. Hier ist wieder die Interpretation unklar: Sollen die Geschwister- oder die Präzedenzbeziehung oder beide negiert werden?

5.1.3 Graphen

Eine Graphbeschreibung setzt sich aus booleschen Ausdrücken über Knotenrelationen, Knotenbeschreibungen, Graphprädikaten (s.u.) und Templateaufrufen (vgl. Abschnitt 5.5) zusammen. Ein Negationsoperator ist hier bewusst ausgeklammert. Denn zum einen ist die unklare Semantik einer negierten Relation (s.o.) eine schwache Grundlage für ein solches Sprachkonstrukt. Und zum anderen schafft der Negationsoperator erhebliche Probleme für eine Verarbeitung von Variablen (vgl. Teil III).

Die folgenden Relationen beschreiben den Teilgraph der Nominalphrase in Abbildung 5.4:

```
            #np:[cat="NP"] &
        #w1:[word="ein" & pos="ART"] &
        #w2:[word="Mann" & pos="NN"] &
(#np >NK #w1) & (#np >NK #w2) & (#w1 . #w2)          (5.19)
```

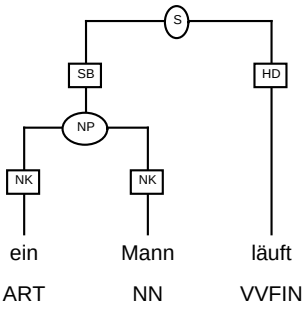


Abbildung 5.4: Ein Beispielgraph

Um (Teil-)Graphen noch genauer beschreiben zu können, sind zusätzlich einige Prädikate vorgesehen. Diese Prädikate werden hier aufgelistet.

Stelligkeit (arity, tokenarity)

Wenn man sich den Graphen in Abbildung 5.4 und die zugehörige Graphbeschreibung (5.19) genau ansieht, so ist festzustellen, dass dieser Ausdruck nach wie vor unterspezifiziert ist. Denn der Ausdruck lässt weiterhin zu, dass noch weitere Knoten zur Nominalphrase gehören.

Um im Hinblick auf die Korpusdefinition zu einer vollständigen Spezifikation zu kommen, wird das Prädikat `arity` verwendet. Es gibt die Anzahl der Töchter eines Knotens an, die sogenannte Stelligkeit. Als Varianten werden die exakte Stelligkeit `arity(#v,n)` (`#v` hat n Töchter) und die Bereichsstelligkeit `arity(#v,m,n)` (`#v` hat mindestens m, aber höchstens n Töchter) unterschieden. Der folgende Ausdruck beschreibt die Nominalphrase in Abbildung 5.4 nun eindeutig:

```
                #np:[cat="NP"] &
          #w1:[word="ein" & pos="ART"] &
          #w2:[word="Mann" & pos="NN"] &
(#np >NK #w1) & (#np >NK #w2) & (#w1 . #w2) &
                  arity(#np,2)                    (5.20)
```

Im Unterschied zur Stelligkeit beschreibt die Token-Stelligkeit die Anzahl der Token, die ein Knoten (beliebig tief) dominiert. Für den Wurzelknoten `#w` in Abbildung 5.4 gilt also beispielsweise `tokenarity(#w,3)`.

Wurzelknoten

Das Prädikat `root(#v)` ist genau dann erfüllt, wenn `#v` Wurzelknoten des zugehörigen Korpusgraphen ist. Dieses Prädikat gibt zur Definition wie auch zur Anfrage ein Hilfsmittel an die Hand, eindeutig auf den Wurzelknoten und damit auch über die Relationen linke und rechte Ecke eindeutig auf den Satzanfang bzw. das Satzende zuzugreifen.

Nützlich ist dieses Prädikat auch in Kombination mit den Stelligkeits-Prädikaten. So beschreibt der folgende Ausdruck Sätze, die maximal 10 Token umfassen:

```
root(#v) & tokenarity(#v,1,10)
```

Es sei an dieser Stelle bemerkt, dass die Festlegung des Wurzelknotens bei der Definition eines Korpusgraphen streng genommen redundant ist, da der Wurzelknoten aus den Dominanzbeziehungen des Graphen berechnet werden könnte. Das Design ist jedoch in sich geschlossener, wenn die Beschreibung eines Korpusgraphen vollständig mit Wurzelknoten spezifiziert ist.

Es sei weiterhin bemerkt, das viele Baumbanken keinen eindeutigen Wurzelknoten in der Graphstruktur vorsehen. Im Falle der TIGER-Baumbank werden beispielsweise die Satzzeichen nicht in den Graphen eingehängt (vgl. etwa Abb. 5.4). In diesem Fall muss die spätere Implementation einen künstlichen Wurzelknoten einfügen, der alle Graphknoten umspannt. Über diese Konvention kommt das Korpus der Forderung nach einem eindeutig bestimmten Wurzelknoten nach.

Kontinuität

Kantenkreuzungen sind eine besondere Eigenschaft des Datenmodells. Wenn Kantenkreuzungen auch zur Annotation von Phänomenen verwendet werden, müssen sie entsprechend abfragbar sein. Im Beispielsatz in Abb. 5.5 ist an der kreuzenden Kante festzumachen, dass es sich hier um einen extraponierten Relativsatz handelt.

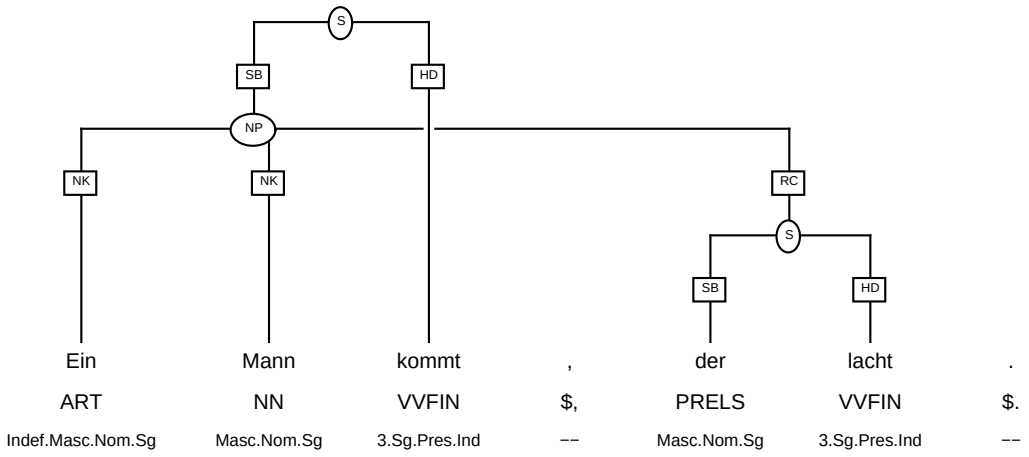


Abbildung 5.5: Annotation eines extraponierten Relativsatzes

Die folgende Definition gibt nun ein Hilfsmittel zur Abfrage an die Hand: Ein durch den Wurzelknoten `#v` repräsentierter Subgraph ist kontinuierlich (`continuous(#v)`), wenn die Projektion des Wurzelknotens auf die Terminalebene kontinuierlich ist. Ansonsten ist dieser Subgraph diskontinuierlich (`discontinuous(#v)`). Eine Projektion ist genau dann kontinuierlich, wenn sie fortlaufende Terminalknoten umfasst.

Für den Beispielsatz lautet die Anfrage nach einem extraponierten Relativsatz:

```
#np:[cat="NP"] & (#np >RC [cat="S"]) & discontinuous(#np)   (5.21)
```

5.2 Korpusdefinition

Eine Korpusdefinition besteht aus einer Liste von eingeschränkten Graphbeschreibungen. Die Einschränkungen bestehen in folgenden Punkten:

1. Attributdeklaration

   Eine zusätzliche Korpusdefinition umfasst die Deklaration aller verwendeten Attribute, ihrer Gültigkeitsbereiche (innere Knoten, äussere Knoten

oder beliebige Knoten) und ihrer Wertebereiche. Diese Deklaration wird für Konsistenzprüfungen, Universumsbildungen und die Korpusdokumentation verwendet (für Details vgl. Unterabschnitt 5.2.1).

2. Grapheigenschaften

   - Es gibt genau einen ausgezeichneten Wurzelknoten.
   - Jeder Knoten außer dem Wurzelknoten wird von genau einem anderen Knoten dominiert.
   - Kein Knoten darf sich selbst dominieren, sei es direkt oder indirekt.

3. Vollständige Disambiguierung

   - Der Wurzelknoten ist für jeden Graphen explizit angegeben.
   - Graphbeschreibungen dürfen nur die Basisrelationen der beschrifteten direkten Dominanz >L und der linearen Präzedenz . umfassen.
   - Die Attributspezifikation eines Knotens besteht aus einer Konjunktion von Attribut-Wert-Paaren. Dabei ist jedes Attribut vertreten, das für diesen Knotentyp (Terminal bzw. Nichtterminal) definiert ist.
   - Für jeden Nichtterminal-Knoten ist seine Stelligkeit explizit angegeben.
   - Die Präzedenzrelationen der Wortknoten bilden eine totale Ordnung.
   - Als einzige logische Verknüpfung auf allen Ebenen der Beschreibungssprache ist die Konjunktion erlaubt.
   - Weder Typen (vgl. Abschnitt 5.3) noch Templates (vgl. Abschnitt 5.5) noch reguläre Ausdrücke dürfen verwendet werden.

Durch die Grapheigenschaften wird insbesondere die Zyklenfreiheit des Graphen sichergestellt. Die Disambiguitätsforderungen lassen nur genau die Graphbeschreibungen zu, die zur eindeutigen Festlegung erforderlich sind.

Zur Trennung von Graphbeschreibungen wird eine XML-ähnliche Umrahmung um jede Beschreibung eines Korpusgraphen vorgenommen. Die Vergabe einer Graph-ID dient der eindeutigen Unterscheidung der Korpusgraphen. Der diskutierte Beispielgraph (vgl. Abb. 5.4) wird durch die nachfolgende Definition (vgl. Abb. 5.6) nun vollständig beschrieben.

5.2.1 Attributdeklaration

Eine umfassende Deklaration aller Attribute hat große Vorteile für die Korpusdefinition, die Korpusanfrage wie auch für die Gestaltung der Benutzeroberfläche:

```
<sentence id="graphid">

"1":[word="ein" & pos="ART"] &
"2":[word="Mann" & pos="NN"] &
"3":[word="läuft" & pos="VVFIN"] &
"4":[cat="NP"] &
"5":[cat="S"] &

"1" . "2" & "2" . "3" &

("5" >SB "4")  &  ("5" >HD "3") &
("4" >NK "1")  &  ("4" >NK "2") &

arity("4",2) & arity("5",2) &

root("5")

</sentence>
```

Abbildung 5.6: Vollständige Spezifikation des Beispielgraphen

- Durch die Deklaration der Attribute können Inkonsistenzen wie das Fehlen eines Attribut-Wert-Paares für einen Knoten oder Tippfehler bei einem Attributwert sofort aufgedeckt werden.
- Die Liste aller Attributwerte wird zur Bildung des Universums verwendet. Das Universum ist u.a. Voraussetzung für die Verarbeitung von negierten Typsymbolen (vgl. Teil III).
- Da die Attributwerte auch mit einer Kurzbeschreibung erläutert werden können, stellt die Deklaration eine hervorragende Korpusdokumentation dar. Sie wird ebenfalls als Eingabehilfe in der Benutzeroberfläche verwendet (vgl. Teil IV).

Das folgende Beispiel zeigt eine vollständige Deklaration für ein nur aus dem Beispielsatz (vgl. Abb. 5.4 und Abb. 5.6) bestehendes Korpus. Für Attribute mit offenem Wertebereich (z.B. Wortform oder Lemma) darf die Aufzählung fehlen. Als Gültigkeitsbereiche für Attribute können terminale Knoten (`T`), nicht-terminale Knoten (`NT`) oder auch beide Bereiche (`FREC` für *feature record*) angegeben werden. Neben den Knotenattributen werden auch Kantenbeschriftungen für Kanten und sekundäre Kanten deklariert. Hier ist die Angabe des Gültigkeitsbereichs nicht erforderlich.

```
<feature name="word" domain="T" />

<feature name="pos" domain="T">
  <value name="ART">Artikel</value>
  <value name="NN">normales Nomen</value>
  <value name="VVFIN">finites Verb</value>
</feature>

<feature name="cat" domain="NT">
  <value name="NP">Nominalphrase</value>
  <value name="S">Satz</value>
</feature>

<edgelabel>
  <value name="HD">Kopf</value>
  <value name="NK">Teil des NP-Kerns</value>
  <value name="SB">Subjekt</value>
</edgelabel>
```

Die Attributdeklaration stellt die Grundlage für die vordefinierten Typen eines Korpus dar. Das verwendete Typsystem wird im folgenden Abschnitt vorgestellt.

5.3 Typen

Das Typsystem hat in der vorliegenden Systemarchitektur zwei Aufgaben. Zum einen vereinfacht es die Anfrageverarbeitung (vgl. Abiteboul et al. 2000, Kapitel 7.1). Inkonsistenzen können bereits im Vorfeld erkannt werden, zum Teil kann in der Verarbeitung mit den Typnamen statt einer Menge von Instanzen gearbeitet werden. Zum anderen kann eine Typisierung ein Korpus bzw. seine Annotation strukturieren und damit für den Anwender leichter zugänglich machen. Für die Beschreibungssprache stehen dazu einige vordefinierte Typen zur Verfügung (vgl. 5.3.1), der vordefinierte Typ `UserDefConstant` für Attributwerte kann zudem vom Benutzer erweitert werden (vgl. 5.3.2).

5.3.1 Vordefinierte Typen

Abb. 5.7 zeigt alle vordefinierten Typen im Überblick. Im Einzelnen sind folgende Typen vorgesehen:

- `String` steht für alle Werte eines Attributs, das in der Attributdeklaration zwar angegeben, dessen Attributwerte aber nicht aufgelistet worden sind

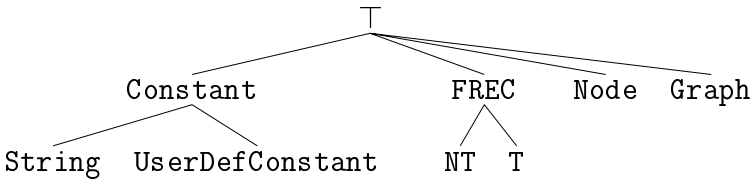


Abbildung 5.7: Hierarchie der vordefinierten Typen

(vgl. 5.2.1). Der Typ `String` ist damit stets mit einem Attribut verbunden:

```
[word = String]
```

(5.22)

- `UserDefConstant` steht für alle Werte eines Attributs, die allesamt in der Deklaration aufgelistet worden sind. Auch der Typ `UserDefConstant` ist damit stets mit einem Attribut verbunden:

  ```
  [pos = UserDefConstant]
  ```

  (5.23)

- `Constant` umfasst die Typen `String` und `UserDefConstant`. Dieser Typ kann unabhängig von der Attributdeklaration verwendet werden. Die folgende Knotenbeschreibung spezifiziert Wortknoten mit beliebiger `word`- und `pos`-Attributbelegung.

  ```
  [word=Constant & pos=Constant]
  ```

  (5.24)

- `NT` steht für alle Attributspezifikationen nicht-terminaler Knoten; `T` steht für alle Attributspezifikationen von terminalen Knoten. Welche Attribute innerhalb `T` bzw. `NT` zulässig sind, wird dabei durch die Attributdeklaration festgelegt (vgl. 5.2.1). Die Knotenbeschreibungen `[NT]` bzw. `[T]` beschreiben alle Nichtterminale bzw. alle Terminale. Im folgenden Beispiel wird der untergeordnete Knoten als Terminal festgelegt:

  ```
  [cat="NP"] >HD [T]
  ```

  (5.25)

  Im folgenden Beispiel matcht die Knotenbeschreibung alle Nichtterminalknoten sowie alle Wortknoten mit der Wortform Haus:

  ```
  [NT | word="Haus"]
  ```

  (5.26)

- `FREC` umfasst die beiden Typen `NT` und `T`. So beschreibt im folgenden Beispiel die erste Knotenbeschreibung einen beliebigen Knoten:

  ```
  [FREC] $ [pos="VVFIN"]
  ```

  (5.27)

- `Node` steht für alle Knotenbeschreibungen.
- `Graph` steht für alle Graphbeschreibungen.

5.3.2 Typdefinition für Attribute

Benutzereigene Typdefinitionen für ein Attribut, d.h. eine Erweiterung des für das Attribut vordefinierten Typs `UserDefConstant`, verhelfen zu einer sprechenderen Formulierung von Attribut-Wert-Paaren. Eine benutzereigene Typdefinition für ein Attribut hat die Aufgabe, die Attributwerte des Attributs zu strukturieren. Sie stellt damit gleichzeitig eine Dokumentation des Attributs dar. Eine Typdefinition erfolgt getrennt vom Korpus, sie wird erst zur Laufzeit an ein Korpus gebunden. Jeder Benutzer kann damit prinzipiell seine eigenen Typdefinitionen für die deklarierten Attribute verwenden.

Für eine Typdefinition gilt die *closed world*-Interpretation, die einen Typ durch eine exhaustive Aufzählung seiner Untertypen beschreibt. Der Einfachheit halber wird ein hierarchisches Typsystem verwendet. *Closed world* bedeutet, dass das Wurzelsymbol der Typhierarchie, das an den vordefinierten Typ `UserDefConstant` gebunden wird, alle in der Korpusdefinition deklarierten Attributwerte direkt oder indirekt über Subtypen umfasst. Damit die Negation eines Typsymbols eindeutig definiert ist, muss jede Attributwert-Konstante an genau einen Typen gebunden sein.

Auf der linken Seite einer Typdefinition steht der Name des zu definierenden Typs. Auf der rechten Seite werden die untergeordneten (Sub-)Typen bzw. Konstanten aufgelistet. Das folgende Beispiel zeigt einen Ausschnitt eines Typsystems für das Attribut `pos` über dem STTS-Tagset (vgl. auch Abb. 5.8):

```
pos := nominal, other;
nominal := adjective, noun, pronoun;
adjective := "ADJA", "ADJD";
noun := "NN", "NE";
pronoun := "PPER", "PREL", ... ;
```

Jeder (Sub-)Typ darf nur genau einmal definiert werden. Zudem muss jeder Subtyp genau einmal von einem übergeordneten Typ eingebettet werden. Durch diese Einschränkungen werden rekursive Definitionen und Mehrfach-Einbettung von Subtypen verhindert. Für Anwendungen, die solche Einbettungen verlangen, stellen Templates eine Ausweichlösung dar (vgl. Abschnitt 5.5).

Die Wurzel des Typsystems ist das zuerst deklarierte nicht-terminale Symbol (hier: `pos`) und wird implizit über eine System-Typdefinition eingebettet:

```
UserDefConstant := pos;
```
(5.28)

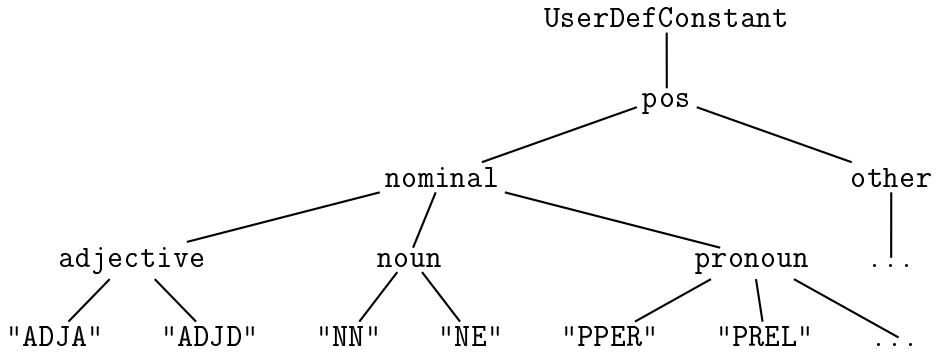


Abbildung 5.8: Ein Typsystem für das Wortarten-Attribut `pos`

Mit Hilfe eines benutzerdefinierten Typsystems können Anfragen nun sehr sprechend formuliert werden:

```
[cat="NP"] > [pos=nominal]                                    (5.29)
```

5.4 Logische Variablen

5.4.1 Vorüberlegungen

Die Ausdruckskraft einer Sprache wird maßgeblich durch die Verfügbarkeit von Variablen bestimmt. Bei der Festlegung des Variablenkonzepts können folgende Konzepte in Betracht gezogen werden:

- Prozedurale Variablen

  Prozedurale Variablen können an einer Stelle definiert und später wieder verwendet werden. Findet eine erneute Definition statt, so wird die erste Definition überschrieben. Im folgenden Beispiel würde die Knotenvariable `#v` zunächst als Nominalphrase, später dann als Verbalphrase besetzt:

  ```
  #v:[cat="NP"] > [pos="NN"] & #v > [pos="NE"] &
          #v:[cat="VP"] > [pos="VVFIN"]
  ```

  Nachteil dieses Konzepts ist die Abhängigkeit des Anfrageergebnisses von der Reihenfolge der Anfragebestandteile (keine Symmetrie boolescher Verknüpfungen) und damit ein erschwertes Nachvollziehen des Ergebnisses durch den Anwender.

- Einmalig spezifizierbare Variablen

  Bei diesem Konzept darf eine Variable nur genau einmal spezifiziert werden. Anschließend kann sie beliebig oft verwendet werden:

```
#v:[cat="NP"] > [pos="ART"] &
#v > [pos="ADJA"] & #v > [pos="NN"]
```

- Logische Variablen

  Wird eine logische Variable mehrmals in einer Anfrage verwendet, so wird die Identität der bezeichneten Strukturen verlangt. Im folgenden Beispiel wird die erste Beschreibung von `#t` mit der zweiten Beschreibung unifiziert.

```
#t:[pos="ART"] & #t:[word="der"]
```

(5.30)

  Dazu werden die an `#t` geknüpften Bedingungen konjunktiv verknüpft. Als Ergebnis der Unifikation entsteht der folgende Ausdruck:

```
#t:[pos="ART" & word="der"]
```

(5.31)

  Dieses Konzept ist für den Anwender bequemer, da er die Spezifikation einer Variablen bei Bedarf aufteilen kann. Sinnvoll ist diese Möglichkeit zum Beispiel beim schrittweisen Verfeinern einer Anfrage:

```
[cat="NP"] > #t:[pos="ART"] & ... & #t:[word="der"]
```

(5.32)

Um dem Anwender möglichst viel Freiheiten zu lassen, werden in der Korpusbeschreibungssprache logische Variablen verwendet. Da das Konzept logischer Variablen dem Konzept der Koreferenz von Merkmalsstrukturen entspricht, wird die Verwendung logischer Variablen im Folgenden anhand von Merkmalsstrukturen illustriert. Man betrachte dazu folgenden Ausdruck der Sprache und die entsprechende Merkmalsstruktur[1]. Die drei Koreferenzboxen zeigen, auf welchen Ebenen Variablen eingesetzt werden können: Knoten-IDs [1], Attributspezifikationen [2] und Attributwerte [3].

```
[cat="NP" & case="ACC"]
```

(5.33)

$$\begin{bmatrix} \text{ID} & \boxed{1}\alpha \\ \text{ATTR} & \boxed{2}\begin{bmatrix} \text{CAT} & \text{NP} \\ \text{CASE} & \boxed{3}\text{ACC} \end{bmatrix} \end{bmatrix}$$

(5.34)

[1] Da innerhalb von Merkmalsstrukturen Attribute mit unspezifiziertem Attributwert einfach weggelassen werden, ist die Angabe des ID-Attributs in diesem Beispiel eigentlich überflüssig. Der Vollständigkeit halber ist hier ein Platzhalter α eingefügt worden. Auf diese Weise bleiben alle nachfolgenden Illustrationen einheitlich.

5.4.2 Attributwerte

Für linguistisch motivierte Anfragen ist auf Attributwert-Ebene das Ausdrücken von Kongruenz zwingend notwendig. Für ein Korpus mit dem Attribut *case* zeigt die folgende Anfrage, wie Variablen auf Attributwert-Ebene eingesetzt werden können:

```
[cat="NP" & case=#k] $ [cat="NP" & case=#k]
```
(5.35)

Diese Art der Verwendung entspricht der Koreferenz von Attributwerten innerhalb von Merkmalsstrukturen: Die zweite Verwendung `#k` verweist auf die erste Verwendung, beide Attributwerte müssen stets identisch sein. Wichtig ist in diesem Zusammenhang die Unterscheidung von Attributen mit koreferenten Werten (token identity) und Attributen mit nur zufällig übereinstimmenden Werten (type identity). Eine Koreferenz drückt eine explizite Strukturteilung aus, die beiden *case*-Attribute teilen sich einen gemeinsamen Wert:

$$\begin{bmatrix} \text{ID} & \alpha \\ \text{ATTR} & \begin{bmatrix} \text{CAT} & \text{NP} \\ \text{CASE} & \boxed{k} \end{bmatrix} \end{bmatrix} \qquad \begin{bmatrix} \text{ID} & \beta \\ \text{ATTR} & \begin{bmatrix} \text{CAT} & \text{NP} \\ \text{CASE} & \boxed{k} \end{bmatrix} \end{bmatrix} \tag{5.36}$$

Wird nun ein Attributwert näher spezifiziert, so gilt diese Spezifikation automatisch für den referierenden Attributwert:

```
[cat="NP" & case=#k:("NOM"|"ACC")] $ [cat="NP" & case=#k]
```
(5.37)

$$\begin{bmatrix} \text{ID} & \alpha \\ \text{ATTR} & \begin{bmatrix} \text{CAT} & \text{NP} \\ \text{CASE} & \boxed{k}\ \text{NOM} \vee \text{ACC} \end{bmatrix} \end{bmatrix} \qquad \begin{bmatrix} \text{ID} & \beta \\ \text{ATTR} & \begin{bmatrix} \text{CAT} & \text{NP} \\ \text{CASE} & \boxed{k} \end{bmatrix} \end{bmatrix} \tag{5.38}$$

In der Beschreibungssprache kann zudem die Spezifikation einer Variablen aufgeteilt werden:

```
[cat="NP" & case=#k:("NOM"|"ACC")] $
 [cat="NP" & case=#k:("NOM"|"DAT")]
```
(5.39)

In diesem Falle werden die Beschreibungen zunächst unifiziert und dann einer Variablen zugeordnet. Falls die beiden Attributwerte nicht unifizierbar sind, so ist die Anfrage inkonsistent. Der aus der Unifikation resultierende Ausdruck sieht in diesem Beispiel folgendermaßen aus:

```
[cat="NP" & case=#k:"NOM"] . [cat="NP" & case=#k]
```
(5.40)

5.4.3 Attributspezifikationen

Die für Attributwerte aufgezeigte Variablenkoreferenz lässt sich nun analog auf die Koreferenz der gesamten Attributspezifikation übertragen. Im folgenden Ausdruck teilen sich die beiden Knoten die gesamte Attributspezifikation:

```
[#f: cat="NP" & case=("NOM"|"ACC")] . [#f]
```
(5.41)

$$\left[\begin{array}{ll}\text{ID} & \alpha \\ \text{ATTR} & \boxed{1}\left[\begin{array}{ll}\text{CAT} & \text{NP} \\ \text{CASE} & \text{NOM} \vee \text{ACC}\end{array}\right]\end{array}\right] \qquad \left[\begin{array}{ll}\text{ID} & \beta \\ \text{ATTR} & \boxed{1}\end{array}\right] \tag{5.42}$$

Man beachte, dass durch die Strukturteilung auch eine Identifikation von Attributen erfolgt, die in dem Ausdruck nicht auftreten. Dies wird an folgendem Beispiel deutlich:

```
[#f: cat="NP"] . [#f]
```
(5.43)

Der Wert des Attributs *case* ist hier nicht festgelegt, demnach unterspezifiziert. Der obige Ausdruck lässt sich also auch folgendermaßen schreiben:

```
[#f: cat="NP" & case=#p] . [#f]
```
(5.44)

Doch durch die Teilung der gesamten Struktur wird auch eine Teilung der Attributwerte impliziert. Demnach müssen auch die Werte für das Attribut *case* übereinstimmen.

Sollten analog zu Variablen auf Attributwert-Ebene koreferente Attributspezifikationen mehr als nur einmal spezifiziert werden, findet zunächst wiederum eine Unifikation der bezeichneten Strukturen (d.h. der Attributwerte) statt.

5.4.4 Knotenbeschreibungen

Wie bereits in 5.1.1 erläutert, stellt eine Knotenvariable einen Platzhalter für eine Knoten-ID dar. Statt der festen Knoten-ID, die zur Knotendefinition gebraucht wird, matcht eine Knotenvariable eine Knoten-ID im Korpus. Im folgenden Ausdruck muss die Variable `#np` durch dieselbe Knoten-ID belegt werden:

```
#np:[cat="NP" & case=#k:("NOM"|"ACC")] & (#np > [pos="NN"])
```
(5.45)

$$\left[\begin{array}{ll}\text{ID} & \boxed{\text{np}} \\ \text{ATTR} & \left[\begin{array}{ll}\text{CAT} & \text{NP} \\ \text{CASE} & \text{ACC}\end{array}\right]\end{array}\right] \qquad \left[\text{ID}\ \boxed{\text{np}}\right] \tag{5.46}$$

Da eine Knoten-ID einen Korpusknoten in eindeutiger Weise bezeichnet, verweist die zweite Verwendung nicht nur auf eine geteilte Attributspezifikation, sondern auf denselben Knoten im Korpus. Als Konsequenz sieht eine adäquate Visualisierung als Merkmalsstruktur folgendermaßen aus:

$$\boxed{\text{np}}\begin{bmatrix}\text{ID} & \alpha \\ \text{ATTR} & \begin{bmatrix}\text{CAT} & \text{NP} \\ \text{CASE} & \text{ACC}\end{bmatrix}\end{bmatrix} \qquad \boxed{\text{np}} \tag{5.47}$$

Sollte die Knotenbeschreibung einer Knotenvariablen mehr als nur einmal spezifiziert sein, findet wieder zunächst eine Unifikation der Attributspezifikationen statt. Beispiel:

```
(#np:[cat="NP"] > [pos="ADJA"]) &
(#np:[cat=("NP"|"VP")] > [pos="NN"])                    (5.48)
```

In diesem Beispiel ensteht als Ergebnis der Unifikation der folgende Ausdruck:

```
(#np:[cat="NP"] > [pos="ADJA"]) & (#np > [pos="NN"])    (5.49)
```

5.4.5 Kantenbeschriftungen

Für Kantenbeschriftungen werden implizit bereits bei der direkten Dominanz Variablen benutzt, da die Relation > die Kantenbeschriftung unterspezifiziert lässt. Eine explizite Verwendung würde jedoch die Ausdruckskraft der Sprache erhöhen und damit die Verarbeitung deutlich komplexer gestalten. Da die Zahl der Kantenbeschriftungen eines Korpus meist gering ist, scheint die disjunktive Aufzählung der Alternativen als Ausdrucksmittel ausreichend. Aus diesem Grunde sind keine Variablen auf Kantenbeschriftungsebene zugelassen.

5.5 Templates

Ein Modularisierungsmechanismus kann die Arbeit mit der Beschreibungssprache enorm vereinfachen. Wiederkehrende Ausdrücke können gekapselt und über einen stellvertretenden Ausdruck nutzbar gemacht werden. Für die TIGER-Beschreibungssprache werden Templates als Relationen zur Verfügung gestellt. Als einzige Einschränkung darf ein Template sich nicht selbst aufrufen, weder direkt noch indirekt. Die Schreibweise und die Verarbeitung orientiert sich an der Programmiersprache Prolog.

Das folgende Beispiel beschreibt Nominalphrasen, die lediglich aus Artikel und Nomen bestehen und deren Kasus parametrisierbar ist:

```
NounPhrase(#np, #k) <-
  #np:[cat="NP"]
  #np > #w1:[pos="ART" & case=#k] &
  #np > #w2:[pos="NN" & case=#k] &
  #w1 . #w2 &
  arity(#np,2);
```

Ein Template ist in Kopf und Rumpf unterteilt. Syntaktisch gesehen werden Kopf und Rumpf durch den Operator <- getrennt und das Ende des Rumpfs durch ein Semikolon markiert. Ein Template umfasst im Templatekopf neben dem Templatenamen eine Liste von Parametern und im Templaterumpf eine Graphbeschreibung.

Templatedefinitionen zu einem Korpus werden separat verwaltet. Technisch gesehen ist beispielsweise die Auflistung in einer Datei vorstellbar. Sollen Alternativen für denselben Templateheader angegeben werden, so werden diese entweder als Disjunktion innerhalb des Templaterumpfs ausgedrückt oder als eigenständige Templates nacheinander angegeben.

Wie auch im Beispiel zu sehen, dürfen die Templateparameter aus unterschiedlichen Beschreibungsebenen stammen. So bezeichnet die Variable `#np` eine Knotenbeschreibung, die Variable `#k` hingegen einen Attributwert. Aus dem Kontext des Templateaufrufs muss dabei die Ebene des Parameters eindeutig erkennbar sein.

In einem Templateaufruf wird nun der Templatekopf angegeben, indem der Templatename mit Parametern der übrigen Anfrage verknüpft wird. Im folgenden Beispiel werden Artikel-Nomen-Nominalphrasen im Nominativ beschrieben.

```
#np:[case=#k:"nom"] & NounPhrase(#np,#k)        (5.50)
```

Ein Templateaufruf ist für sich allein mit einer leeren Graphbeschreibung assoziiert. Die Semantik des Templateaufrufs entspricht also allen Graphbeschreibungen des Korpus. Erst durch die weitere Verwendung eines Parameters in einem Ausdruck bekommt der Ausdruck die gewünschte Semantik (vgl. in Bsp. 5.50: `#np`).

Es sei an dieser Stelle angemerkt, dass Templates zwar Bestandteil der Anfragesprache sind, die Konzeption des Verarbeitungskonzepts und dessen Implementierung aber noch nicht abgeschlossen sind. Sie werden deshalb auch bei der Beschreibung der Anfrageverarbeitung im dritten Teil dieser Arbeit nicht berücksichtigt.

5.6 Korpusanfrage und Anfrageergebnis

Eine Korpusanfrage ist eine beliebige Graphbeschreibung. Ein Anfrageergebnis ist die Menge aller Korpusgraphen, die einen Subgraph enthalten, der die

Anfrage erfüllt. Man beachte, dass ein Korpusgraph in der Regel einen Satz umfasst. Bei Korpora, die gesprochene Sprache beschreiben, wird das Korpus auch durch andere Einheiten strukturiert (vgl. *turns* für die VerbMobil-Korpora in Hinrichs et al. 2000).

Für die Visualisierung von Anfrageergebnissen wird die Ergebnisdefinition noch enger gefasst, da hier die Belegungen der Variablen für den Benutzer von großem Interesse sind. Für die formale Definition der Sprache (vgl. Kapitel 6) und die Darstellung des Verarbeitungskonzepts (vgl. Teil III) stellt die graphenweise Sicht jedoch eine sinnvolle Vereinfachung dar.

5.7 Erweiterungen

Ein Allquantor würde die Mächtigkeit der Sprache enorm vergrößern. Zur Diskussion dieses Sprachmittels betrachte man folgenden Beispielausdruck:

```
[cat="VP"] >* [cat=!"NP"]                                    (5.51)
```

Auf den ersten Blick mag für diesen Ausdruck vermutet werden, dass sich unter der gesuchten Verbalphrase **keine** Nominalphrase befindet. Tatsächlich wird aber hier ausgedrückt, dass **mindestens eine** Nicht-Nominalphrase von der Verbalphrase dominiert wird. Die beiden Knotenbeschreibungen sind also durch einen Existenzquantor gebunden:

```
∃#X ∃#Y:  #X:[cat="VP"] >* #Y:[cat=!"NP"]                    (5.52)
```

Falls hingegen ausgedrückt werden soll, dass sich unter der gesuchten Verbalphrase keine Nominalphrase befindet, ist ein Allquantor und der Implikationsoperator notwendig:

```
∃#X ∀#Y:  (#X:[cat="VP"] >* #Y) ⇒ (#Y:[cat=!"NP"])           (5.53)
```

Für die Einführung eines Allquantors spricht, dass die Ausdruckskraft der Sprache enorm erhöht wird.

Auf der anderen Seite kann ein Allquantor zu Laufzeitproblemen führen, da die Auswertung kostspielig ist. Zudem werden in der Anfragesprache drei Ebenen der Variablenverwendung betrachtet, was die Komplexität zusätzlich erhöht. Die Integration des Sprachkonstrukts wird die Anfragesprache außerdem unübersichtlicher machen.

Als Ergebnis dieser Diskussion ist in einer ersten Version der Anfragesprache und der zugehörigen Implementierung der Anfrageauswertung zunächst auf einen Allquantor verzichtet worden. Ein Konzept zur Realisierung eines eingeschränkten Allquantors wird im Ausblick dieser Arbeit skizziert (vgl. Abschnitt 15.2).

5.8 Zusammenfassung

Das vorliegende Kapitel hat die Konzeption der Korpusbeschreibungssprache aufgezeigt. In wie weit diese Konzeption den zuvor formulierten Anforderungen gerecht wird, ist nicht objektiv zu entscheiden. Hier sind Usability-Studien erforderlich, in denen die Anwender die Qualität der Konzeption beurteilen.

Kapitel 6

Syntax und Semantik

Dieses Kapitel dient der formalen Definition der TIGER-Korpusbeschreibungssprache. In einem ersten Schritt wird das zugrunde liegende Datenmodell formalisiert. Anschließend werden Syntax und Semantik der Sprache festgelegt.

Das Kapitel erfüllt zwei Aufgaben: Zum einen dient es der Transparenz gegenüber dem Anwender des Suchwerkzeugs, da die Syntax der Sprache und die Semantik der Sprachbestandteile formal definiert werden. Bei Unklarheiten bzgl. der Semantik eines Sprachkonstrukts kann dieses Kapitel zu Rate gezogen werden. Zum anderen stellt das Kapitel eine Spezifikation dar, an die die spätere Implementation gebunden ist. Abweichungen der Semantik der Implementation sollen damit verhindert werden.

Das vorliegende Kapitel stellt die Zusammenfassung eines Arbeitsberichts dar (König und Lezius 2002b), der sowohl die Syntax und Semantik der Sprache als auch den verwendeten Verarbeitungskalkül definiert.

6.1 Das Syntaxgraph-Datenmodell

Das verwendete Datenmodell wird als Syntaxgraph-Modell bezeichnet. Der Begriff Syntaxgraph deutet an, dass (schwache) gerichtete Graphen zur syntaktischen Beschreibung von Korpuseinheiten verwendet werden. Um jederzeit Generalisierungen des Datenmodells nachtragen zu können, wird das Datenmodell so allgemein definiert, dass die Struktur jedes Bestandteils erweitert werden kann.

Das Datenmodell macht keine Aussagen über konkrete Attribute, Typen, Templates o.ä., da diese erst mit der Korpusdefinition feststehen. Sekundäre Kanten werden bei der Formalisierung des Datenmodells ausgeklammert. Sie bilden eine separate Schicht des Modells und werden auch bei der Anfrageverarbeitung in einer separaten Schicht realisiert. Kantenbeschriftungen werden für ein Korpus stets vorausgesetzt. Die Implementation wird aber auch mit Korpora umgehen können, die auf Kantenbeschriftungen verzichten (vgl. Teil III).

Ein Syntaxgraph-Modell besteht aus einer Menge von Syntaxgraphen. Ein Syntaxgraph G stellt eine Relation über einer Menge von Graphknoten V dar. Während aus graphentheoretischer Sicht Knoten die elementaren Einheiten eines Graphen darstellen, werden an den Knoten eines Syntaxgraphen noch weitere Informationen abgelegt.

6.1.1 Attributtabellen

Da Attribute einer allgemeinen Merkmalsstruktur als Werte wieder eine Merkmalsstruktur enthalten dürfen, die Wertebereiche der hier betrachteten Attribute aber auf Konstanten eingeschränkt sind, sprechen wir hier nicht von Merkmalsstrukturen, sondern von Attributtabellen. Dieser Begriff soll andeuten, dass jedem Attributname ein Attributwert zugeordnet wird.

Definition 1 (Konstanten)
Mit C sei eine nicht-leere Menge von Konstanten bezeichnet.

Definition 2 (Attributtabelle)
Eine Attributtabelle F ist eine Menge von Zwei-Tupeln $\langle f_i, c_i \rangle$, wobei f_i den Attributnamen und $c_i \in C$ den Attributwert bezeichnet. Für zwei Elemente von F sind die Attributnamen jeweils verschieden, d.h. für $l_i = \langle f_i, c_i \rangle \in F$ und $l_j = \langle f_j, c_j \rangle \in F$ gilt: $l_i \neq l_j \Rightarrow f_i \neq f_j$.
Die Menge aller Attributtabellen wird als $\mathcal{F}$ bezeichnet.

Definition 3 (Attribut)
Ein Attribut $f_i : \mathcal{F} \rightarrow C$ ist eine Abbildung, die einer Attributtabelle den Attributwert des gleichnamigen Attributnamens zuordnet, d.h. $f_i(\{\langle f_1, c_1 \rangle, \ldots, \langle f_i, c_i \rangle, \ldots, \langle f_n, c_n \rangle\}) = c_i$.

6.1.2 Knoten

Definition 4 (Knoten)
Ein Knoten $v \in V$ ist ein Paar $v = \langle \nu, F \rangle$, bestehend aus einer Knoten-ID $\nu \in C$ und einer Attributtabelle F.

Bemerkung 1
Da die Attributtabellen zweier Knoten übereinstimmen können, werden die Graphknoten erst mit der zusätzlichen ID eindeutig unterscheidbar.

6.1.3 Syntaxgraphen

Die folgende Definition legt nun die Struktur eines Syntaxgraphen $G \in \mathcal{G}$ fest. $\mathcal{G}$ bezeichnet die Menge aller Syntaxgraphen.

Definition 5 (Syntaxgraph)
Ein Syntaxgraph G ist ein Sextupel $G = (V_{NT}, V_T, L_G, E_G, O_G, R_G)$ mit folgenden Eigenschaften:

1. *V_{NT} ist die Menge nicht-terminaler oder innerer Knoten. V_{NT} kann auch leer sein.*
2. *V_T ist eine nicht-leere Menge terminaler oder äußerer Knoten.*

   Unter $V_G = V_{NT} \cup V_T$ wird die Menge aller Graphknoten von G verstanden.
3. *L_G ist eine Menge von Kantenbeschriftungen, die auch leer sein darf.*
4. *E_G bezeichnet die beschrifteten, gerichteten Kanten von G. E_G ist eine Menge von Relationen $l \in \langle V_{NT}, (V_{NT} \cup V_T)\rangle$. Ist die Menge der Kantenbeschriftungen nicht leer, so stammen die Namen l der Relationen aus L_G.*
5. *O_G ist eine bijektive Abbildung $O_G : V_T \to \{1, \ldots, |V_T|\}$, die die terminalen Knoten anordnet.*

   D.h. gemäß dieser Abbildung kann für jedes Paar von Terminalknoten $v_i, v_j \in V_T$, $v_i \neq v_j$, eindeutig entschieden werden, ob $O_G(v_i) < O_G(v_j)$ oder $O_G(v_i) > O_G(v_j)$ gilt.
6. *$R_G \in (V_{NT} \cup V_T)$ ist der Wurzelknoten.*

G ist ein Graph mit folgenden Eigenschaften:

1. *G ist ein gerichteter azyklischer Graph mit genau einem Wurzelknoten R_G. Ein Wurzelknoten ist ein Knoten ohne eingehende Kante.*
2. *In alle Knoten $v \in V_G$ mit Ausnahme des Wurzelknotens R_G führt genau eine eingehende Kante.*
3. *Jeder innere Knoten $v \in V_{NT}$ muss mindestens einen Nachfolger besitzen d.h. $v \in V_{NT} \Leftrightarrow \exists l \in E_G$ und $v' \in V_G$ mit $\langle v, v'\rangle \in l$.*

Bemerkung 2
Ein Graph, der lediglich aus einem Terminalknoten $v \in V_T$ besteht, stellt bereits einen vollständigen Syntaxgraphen dar. Man beachte aber, dass zwei oder mehr Terminalknoten allein keinen Syntaxgraphen bilden, da hier der eindeutig bestimmte Wurzelknoten fehlt.

Bemerkung 3
Um analog zu Knoten auch Graphen in eindeutiger Weise voneinander unterscheiden zu können, wird die ID des eindeutigen Wurzelknotens als ID des Graphen definiert.

Bemerkung 4

Im vorausgegangenen Kapitel ist alternativ zum Begriff des terminalen Knotens auch von Wortknoten gesprochen worden. Dieser Begriff ist hier nicht verwendbar, da er bereits linguistisches Wissen über Baumbanken voraussetzt: In der Regel umfasst ein Korpus an den Terminalknoten ein Attribut, das die Wortform enthält.

6.1.4 Begriffsdefinitionen

Im vorliegenden Abschnitt werden Begriffe definiert, die zur Beschreibung von Syntaxgraphen bzw. ihrer Bestandteile verwendet werden.

Definition 6 (Pfad)

Ein Pfad $p = \langle v_0, \ldots, v_n \rangle$ in einem Syntaxgraphen G ist ein n-Tupel von Knoten mit $\langle v_0, v_1 \rangle \in l$ für ein l, wobei entweder $v_1 = v_n$ gilt oder $p' = \langle v_1, \ldots, v_n \rangle$ ein Pfad in G mit $0 < n$ ist. Die Zahl n wird als Länge des Pfads p bezeichnet.

Definition 7 ($>n$, Dominanz in n Stufen, $n > 0$)

Ein Knoten v_1 dominiert einen Knoten v_2 in n Stufen ($v_1 >n\ v_2$), falls es einen Pfad p der Länge n in G gibt mit $p = \langle v_1, \ldots, v_2 \rangle$.

Definition 8 ($>*$, Dominanz)

Ein Knoten v_1 dominiert einen Knoten v_2, falls es einen Pfad p in G gibt mit $p = \langle v_1, \ldots, v_2 \rangle$.

Definition 9 ($>$, direkte Dominanz)

Ein Knoten v_1 dominiert einen Knoten v_2 direkt, falls es einen Pfad p der Länge 1 in G gibt mit $p = \langle v_1, v_2 \rangle$.

Definition 10 ($> @l$, linke Ecke)

Ein Terminalknoten $v_2 \in V_T$ ist die linke Ecke bzgl. eines Knotens v_1 ($v_1 > @l\ v_2$), falls gilt:

- $v_2 = v_1$ *oder*
- $v_1 > *v_2$ *und für alle Knoten $v_i \in V_T$ mit $v_1 > * v_i$ gilt: $O_G(v_2) < O_G(v_i)$.*

Definition 11 ($> @r$, rechte Ecke)

Ein Terminalknoten $v_2 \in V_T$ ist die rechte Ecke bzgl. eines Knotens v_1 ($v_1 > @r\ v_2$), falls gilt:

- $v_2 = v_1$ *oder*
- $v_1 > *v_2$ *und für alle Knoten $v_i \in V_T$ mit $v_1 > * v_i$ gilt: $O_G(v_2) > O_G(v_i)$.*

Bemerkung 5

Wie im vorausgegangenen Kapitel bereits dargelegt (vgl. Unterabschnitt 5.1.2), wird die lineare Präzedenz zweier Knoten auf die Präzedenz ihrer linkesten Nachfahren (linke Ecken) zurückgeführt. Da die Terminalknoten eines Syntaxgraphen per Definition einer totalen Ordnung unterworfen sind, ist die darauf definierte Präzedenz zweier Terminalknoten immer definiert.

Definition 12 ($.n$, Präzedenz mit Abstand n, $n > 0$)

Ein Knoten v_1 geht einem Knoten v_2 mit Abstand n voraus ($v_1 \,.n\, v_2$), falls $n > 0$ und für die linken Ecken v_3 von v_1 ($v_1 > @l\ v_3$) und v_4 von v_2 ($v_2 > @l\ v_4$) gilt: $n = O_G(v_4) \Leftrightarrow O_G(v_3)$.

Definition 13 ($, Geschwister)

Zwei Knoten v_1 und v_2 sind Geschwister ($v_1 \,\$\, v_2$), falls es einen Knoten v_0 gibt, der beide Knoten v_1 und v_2 direkt dominiert, d.h. $\exists\, v_0 \in V_{NT} : \ v_0 > v_1 \ \wedge\ v_0 > v_2 \ \wedge\ v_1 \neq v_2$.

Definition 14 (Stelligkeit n eines Knotens)

Die Stelligkeit n eines Knotens v mit ID ν ist die Anzahl n der ausgehenden Kanten, d.h.

$arity(\nu) = |\ \{v_l | \langle v, v_l \rangle \in l\ \}\ |$.

Definition 15 (Tokenstelligkeit n eines Knotens)

Die Tokenstelligkeit n eines Knotens v mit ID ν ist die Anzahl n der Terminalknoten, die von v dominiert werden, d.h.

$tokenarity(\nu) = |\ \{v_l | \langle v, \ldots, v_l \rangle$ *ist ein Pfad in G und* $v_l \in V_T\ \}\ |$.

Definition 16 (Kontinuierlicher Subgraph mit Wurzel v)

Ein Subgraph mit Wurzelknoten v und ID ν heißt kontinuierlich, falls seine Terminalknoten einen durchgehenden Strom bilden, d.h. für alle von v dominierten Terminalknoten $v_i = v_1, \ldots, v_n$ gilt:

($\exists v_j\ O_G(v_j) = O_G(v_i) + 1$) oder ($\neg\exists v_j\ O_G(v_j) > O_G(v_i)$)

Ansonsten heißt der Subgraph diskontinuierlich.

6.1.5 Relationen

Um auch Typen und Templates modellieren zu können, werden zusätzlich noch Relationen benötigt, die über Konstanten, Attributtabellen, Knoten und Syntaxgraphen definiert werden können. Typen können dann als unäre Relation, d.h. als Prädikate, aufgefasst werden.

Es bezeichne $\mathcal{U}$ das Universum aller Syntaxgraphen und ihrer Substrukturen, d.h. die disjunkte Vereinigungsmenge aller Syntaxgraphen $\mathcal{G}$, Konstanten C, Attributtabellen $\mathcal{F}$ und Knoten V.

Definition 17 (Syntaxgraph Relation)
Eine n-stellige Relation $r \in R$ ist ein n-Tupel ($n > 0$) $r \subseteq \mathcal{U} \times \ldots \times \mathcal{U}$, wobei die Teilmengen von $\mathcal{U}$ nicht leer sein dürfen.

6.1.6 Syntaxgraph-Modell

Nun stehen alle Hilfsmittel zur Verfügung, um die Syntax und Semantik der Beschreibungssprache anhand des Syntaxgraphen-Modells zu definieren. Die folgende Definition fasst alle bisher definierten Komponenten zu einer Einheit zusammen.

Definition 18 (Syntaxgraph-Modell)
Ein Syntaxgraph-Modell ist ein Quintupel $\mathcal{M} = (C, \mathcal{F}, V, \mathcal{G}, R)$.

6.2 Syntax und Interpretation

In diesem Abschnitt wird zum einen die Syntax der TIGER-Beschreibungssprache genau festgelegt, zum anderen dabei auch die semantische Interpretation der Sprache definiert. Die semantische Interpretation bezieht sich dabei stets auf ein gegebenes Syntaxgraph-Modell $\mathcal{M}$ (vgl. Abschnitt 6.1). Es wird hier eine so genannte denotationelle Semantik definiert, d.h. einem zunächst syntaktisch definierten Konstrukt der Sprache wird seine semantische Entsprechung zugeordnet – die so genannte Denotation (Schreibweise $[\![\cdot]\!]$).

Die Interpretation wird auf drei Ebenen beschrieben:

1. Denotation der *Elementarsymbole*
2. Denotation *lokaler Beschreibungen*, d.h. von Attributwerten, Attributspezifikationen, Knotenbeschreibungen und Graphbeschreibungen.

   Anschließend wird der Erfüllbarkeitsbegriff für lokale Beschreibungen definiert.
3. Interpretation *globaler Definitionen*, d.h. der Korpusdefinition (die neben dem Korpus auch die Attributdeklaration enthält), der Template- und Typdefinitionen.

6.2.1 Elementarsymbole

Definition 19 (Elementarsymbole)
Das Inventar der Elementarsymbole setzt sich aus folgenden Symbolen zusammen:

Operatoren

= *(Attributwert)*

! *(Negation)* & *(Konjunktion)* | *(Disjunktion),*

> *(direkte Dominanz),* >* *(allgemeine Dominanz),* >@l *(linke Ecke),* >@r *(rechte Ecke),*

. *(lineare Präzedenz),* .* *(allgemeine Präzedenz),*

$ *(Geschwister),* $.* *(Geschwister mit Präzedenz)*

:= *(Typdefinition)*

Markup

;, <, >, /, domain, feature, value, name

Konstanten c1, c2, ... ∈ **C**

Attributnamen f1, f2, ... ∈ **F**

Kantenbeschriftungen l1, l2, ... ∈ **L**

Typnamen t1, t2, ..., tn, Constant, FREC, Node, Graph, ⊤ *(Top)* ∈ **T**,

sowie ⊥ *(Bottom oder Clash).*

Prädikatnamen root, arity, tokenarity, continuous, discontinuous

Templatenamen r1, r2, ... ∈ **R**

Bemerkung 6 (Notation)
Zur Unterscheidung der Elementarsymbole werden Konstanten stets in doppelte Anführungsstriche "..." eingeschlossen. Typsymbole dürfen nicht mit einem Anführungsstrich beginnen, womit die Mengen **C** *und* **T** *disjunkt sind. Um Verwechslungen mit Variablen zu vermeiden (s.u.), dürfen Konstanten und Typsymbole zudem nicht mit dem Variablen-Symbol # beginnen.*

Bemerkung 7 (Benutzerdefinierte Typsymbole)
Es sei bemerkt, dass die Typsymbole t1, t2, ..., tn *die benutzerdefinierten Typsymbole einschließen.*

Zur Vereinfachung der weiteren Beschreibung sei jedem Elementarsymbol ein semantisches Gegenstück im gegebenen Syntaxgraph-Modell $\mathcal{M}$ zugeordnet. Aus Gründen der Lesbarkeit wird dann folgende vereinfachende Schreibweise verwendet: Statt $c_{\mathcal{M}} \in C_{\mathcal{M}}$ wird kurz $c \in C$ geschrieben.

Definition 20 (Denotation der Elementarsymbole)

Konstanten *Für jede Konstante* `c` $\in$ `C` *gibt es genau eine Konstante* $c \in C$.

Attributnamen *Für jedes Attribut* `f` $\in$ `F` *gibt es genau eine Attributfunktion* $f \in \mathcal{F}$.

Kantenbeschriftungen *Für jede Kantenbeschriftung* `l` $\in$ `L` *gibt es genau eine Kantenbeschriftung* $l \in L$.

Typnamen *Für jedes Typsymbol* `t` $\in$ `T` *gibt es genau eine unäre Syntaxgraph-Relation* $r_{\texttt{t}} \in R$.

Insbesondere gilt $r_{\texttt{Constant}} = C$, $r_{\texttt{FREC}} = \mathcal{F}$, $r_{\texttt{Node}} = V$, $r_{\texttt{Graph}} = \mathcal{G}$, $r_\top = \mathcal{U}$.

Weiterhin gilt $r_\bot = \emptyset$.

Templatenamen *Für jedes* n*-stellige Template* `r` $\in$ `R` *gibt es genau eine Syntaxgraph-Relation* $r \in R$.

Zur Vervollständigung wird nun ein allgemeiner Variablenbegriff definiert.

Definition 21 (Variablensubstitution)
Sei **X** *eine Menge von Variablennamen (im Folgenden stets kurz als Variablen bezeichnet) mit* `#x1`, `#x2`, ... $\in$ **X**. *Eine Variablensubstitution* $\sigma : \mathbf{X} \to \mathcal{U}$ *ordnet einer Variablen ein Element des Universum* $\mathcal{U}$ *zu (genauer eine Knotenbeschreibung, eine Attributspezifikation oder einen Attributwert; vgl. folgende Abschnitte). Der Funktionswert* σ`(#x)` *beschreibt das Element, das der Variablen* `#x` *zugeordnet ist.*

Bemerkung 8 (Variablennamen auf den drei Variablenebenen)
Die Mengen der Variablennamen für Knotenbeschreibungen, Attributspezifikationen und Attributwerte (vgl. folgende Abschnitte) müssen paarweise disjunkt sein. Diese Forderung macht auch in der Praxis Sinn, da sonst die Übersicht über die Wertebereiche der Variablen verloren geht.

Bemerkung 9 (Denotation bzgl. Variablensubstitution)
Als Folge der Definition des allgemeinen Variablenbegriffs muss die Denotation im weiteren Verlauf stets bezüglich einer gegebenen Variablensubstitution betrachtet werden. Zur Vereinfachung wird aber statt der ausführlichen Schreibweise $[\![\gamma]\!]_{\mathcal{M},\sigma'}$ *(Denotation von* γ *im Modell* $\mathcal{M}$ *bezüglich der Variablensubstitution* σ*')* *kurz* $[\![\gamma]\!]$ *geschrieben.*

Die weitere Beschreibung durchläuft die Konstrukte der Korpusbeschreibungssprache von innen nach außen. Dabei wird die Denotation eines Konstrukts einer höheren Stufe auf die Denotation der enthaltenen Konstrukte niedrigerer Stufen zurückgeführt. Die Syntaxdefinition erfolgt in Form von Produktionsregeln.

6.2.2 Attributwerte

Attributwert `d` ohne Variablenpräfigierung

Ein Attributwert-Ausdruck `d` wird als Teilmenge der Menge aller Konstanten C interpretiert. Ein Typ `t` wird als einstellige Relation $r_{\mathtt{t}}$ interpretiert, dessen Argument eine Konstante sein muss. D.h. für jede Konstante ist eindeutig feststellbar, ob sie zu einem Typ zugehörig ist oder nicht.

Syntax		*Erläuterung*	*Denotation* $[\![\cdot]\!]$
`d` $\Rightarrow$	`c`	Konstante	$\{ c \}$
|	`t`	Typ	$\{ u \mid \langle u \rangle \in r_{\mathtt{t}} \} \cap C$
|	`! d`	Negation	$C \setminus [\![\mathtt{d}]\!]$
|	`d & d`	Konjunktion	$[\![\mathtt{d}_1]\!] \cap [\![\mathtt{d}_2]\!]$
|	`d | d`	Disjunktion	$[\![\mathtt{d}_1]\!] \cup [\![\mathtt{d}_2]\!]$

Attributwert `d'` mit Variablenpräfigierung

Variablen dürfen nur an Stelle von Attributwerten oder als Präfix eines Attributwert-Ausdrucks gebraucht werden. Auch Variablen werden wieder als Teilmenge aller Konstanten interpretiert.

Syntax		*Erläuterung*	*Denotation* $[\![\cdot]\!]$
`d'` $\Rightarrow$	`d`	Attributwert	$[\![\mathtt{d}]\!]$
|	`#x`	Variable	$\{ \sigma(\mathtt{\#x}) \} \cap C$
|	`#x:d`	Variable, eingeschränkt durch `d`	$\{ \sigma(\mathtt{\#x}) \} \cap [\![\mathtt{d}]\!]$

6.2.3 Attributspezifikationen

Es folgt die Festlegung der Syntax und Denotation einer *Attributspezifikation* `k'`. Als Ergänzung zu den im vorausgegangenen Abschnitt definierten Attributwerten werden zusätzlich auch reguläre Ausdrücke über Attributwerten als Zeichenketten zugelassen. Ein regulärer Ausdruck darf jedoch weder von einer Variablen referenziert werden noch darf er auf Attributwertebene in einen booleschen Ausdruck eingebettet werden. Bei der Denotation wird eine Funktion `eval` voraus gesetzt, die für einen regulären Ausdruck die Menge aller (als Zeichenkette) matchenden Konstanten zurückliefert. Diese Sichtweise ist näher an der späteren Kalküldefinition bzw. Implementation und erleichtert daher die spätere Bezugnahme auf die Semantik regulärer Ausdrücke.

Attributspezifikationen `k` ohne Variablenpräfigierung

Eine Attributspezifikation wird als Teilmenge aller Attributtabellen $\mathcal{F}$ interpretiert. Im Falle von Attributspezifikationen werden Typen als einstellige Relationen interpretiert, deren Argument eine Konstante ist. D.h. für jede Konstante ist eindeutig entscheidbar, ob sie zu einem Typ zugehörig ist oder nicht.

Syntax		*Erläuterung*	*Denotation* $[\![\cdot]\!]$
`k` $\Rightarrow$	`t`	Typ	$\{ u \mid \langle u \rangle \in r_{\texttt{t}} \} \cap \mathcal{F}$
|	`f = d'`	Attribut-Wert-Paar	$\{F \mid f(F) \in [\![\texttt{d'}]\!]\}$
|	`f != d`	negiertes Attribut-Wert-Paar	$[\![\texttt{f = ! d}]\!]$
|	`f = /z/`	regulärer Ausdruck	$\{F \mid f(F) \in \texttt{eval(z)}\}$
|	`f != /z/`	negierter regulärer Ausdruck	$\{F \mid f(F) \notin \texttt{eval(z)}\}$
|	`!` k	Negation	$F \setminus [\![\texttt{k}]\!]$
|	`k & k`	Konjunktion	$[\![\texttt{k}_1]\!] \cap [\![\texttt{k}_2]\!]$
|	`k | k`	Disjunktion	$[\![\texttt{k}_1]\!] \cup [\![\texttt{k}_2]\!]$

Attributspezifikationen `k'` mit Variablenpräfigierung

Analog zu Attributwerten können auch Attributspezifikationen durch eine Variable präfigiert werden. Syntaktisch werden Attributspezifikationen durch eckige Klammern eingerahmt.

Syntax		*Erläuterung*	*Denotation* $[\![\cdot]\!]$
`k'` $\Rightarrow$	`[ ]`	uneingeschränkt	$\mathcal{F}$
|	`[ k ]`	Attributspezifikation	$[\![\texttt{k}]\!]$
|	`[ #x ]`	Variable	$\{ \sigma(\texttt{\#x}) \} \cap \mathcal{F}$
|	`[ #x:k ]`	Variable, eingeschränkt durch Attributspezifikation `k`	$\{ \sigma(\texttt{\#x}) \} \cap [\![\texttt{k}]\!]$

6.2.4 Knotenbeschreibungen

Eine *Knotenbeschreibung* besteht aus einer Knoten-ID `n` und einer (ggf. präfigierten) Attributspezifikation `k'`, wobei eine der beiden Angaben auch fehlen darf.

Knoten-IDs `n`

Eine Knoten-ID wird als einelementige Menge von Konstanten interpretiert.

Syntax			*Erläuterung*	*Denotation* $[\![\cdot]\!]$
`n`	$\Rightarrow$	`c`	Knoten-ID-Konstante	$\{ c \}$
	|	`#x`	Knoten-ID-Variable	$\{ \sigma(\texttt{\#x}) \} \cap C$

Knoten-Beschreibungen `v`

Eine Knotenbeschreibung wird als Menge von Knoten interpretiert.

Syntax			*Erläuterung*	*Denotation* $[\![\cdot]\!]$
`v`	$\Rightarrow$	`n`	uneingeschränkt	$\{ \langle c, F \rangle \mid c \in [\![\texttt{n}]\!] \}$
	|	`k'`	unidentifiziert	$\{ \langle c, F \rangle \mid F \in [\![\texttt{k'}]\!] \}$
	|	`n:k'`	Knotenbeschreibung	$\{ \langle c, F \rangle \mid F \in [\![\texttt{k'}]\!]$ und $c \in [\![\texttt{n}]\!] \}$

6.2.5 Graphbeschreibungen

Eine *Graphbeschreibung* wird naheliegenderweise als eine Menge von Graphen G interpretiert. Die Syntax und Denotation von Graphbeschreibungen ist in Abb. 6.1 zu finden.

Für Templates wird an dieser Stelle die Syntax des Templateaufrufs, also die Syntax des Templatekopfs, definiert. Dabei werden die Argumente und der eigentliche Aufruf getrennt behandelt.

Argumente `u` eines Templateaufrufs

Syntax			*Erläuterung*	*Denotation* $[\![\cdot]\!]$
`u`	$\Rightarrow$	`#x`	Argumentparameter	$\sigma(\texttt{\#x})$
	|	`#x, u`	Argumente	

Templateaufruf `r'`

Syntax			*Erläuterung*	*Denotation* $[\![\cdot]\!]$
`r'`	$\Rightarrow$	`r(u)`	Templateaufruf	$\begin{cases} \mathcal{G} & \textit{falls } \langle \sigma(x_1), \ldots, \sigma(x_n) \rangle \in r \\ \emptyset & \textit{sonst} \end{cases}$

Wie bereits bei der informellen Erläuterung der Anfragesprache ausgeführt, entspricht die Semantik eines Templateaufrufs für sich allein – sofern der Templateaufruf mit den Variablenbelegungen gültig ist – allen Graphbeschreibungen des Korpus. Ansonsten wird die leere Menge als Semantik festgelegt.

Syntax		*Erläuterung*	*Denotation* $[\![\cdot]\!]$
g $\Rightarrow$	`root(n)`	Wurzelknoten-Identifikation	$\{G \mid [\![\mathtt{n}]\!] = R_G\}$
|	`arity(n,`$\mathtt{i}_1$`)`	Stelligkeit	$\{G \mid arity([\![\mathtt{n}]\!]) = \mathtt{i}_1\}$
|	`arity(n,`$\mathtt{i}_1$`,`$\mathtt{i}_2$`)`	Stelligkeit als Intervall	$\{G \mid \mathtt{i}_1 \leq arity([\![\mathtt{n}]\!]) \leq \mathtt{i}_2\}$
|	`tokenarity(n,`$\mathtt{i}_1$`)`	Tokenstelligkeit	$\{G \mid tokenarity([\![\mathtt{n}]\!]) = \mathtt{i}_1\}$
|	`tokenarity(n,`$\mathtt{i}_1$`,`$\mathtt{i}_2$`)`	Tokenstelligkeit als Intervall	$\{G \mid \mathtt{i}_1 \leq tokenarity([\![\mathtt{n}]\!]) \leq \mathtt{i}_2\}$
|	`continuous(n)`	Kontinuität	$\{G \mid continuous([\![\mathtt{n}]\!])\}$
|	`discontinuous(n)`	Diskontinuität	$\mathcal{G} \setminus \{G \mid continuous([\![\mathtt{n}]\!])\}$
|	`r'`	Templateaufruf	(siehe Unterabschnitt 6.2.5)
|	`v`	Knotenbeschreibung	$\{G \mid [\![\mathtt{v}]\!] \subseteq V_G\}$
|	`v >l v`	beschriftete direkte Dominanz	$\{G \mid l(v_1) = v_2\}$
|	`v > v`	direkte Dominanz	$\{G \mid v_1 >1\ v_2\}$
|	`v >* v`	Dominanz	$\{G \mid \exists i\,(v_1 >i\ v_2)\}$
|	`v >`$\mathtt{i}_3$ `v`	Dominanz mit der Distanz $\mathtt{i}_3$	$\{G \mid v_1 > \mathtt{i}_3\ v_2\}$
|	`v >`$\mathtt{i}_3$`,`$\mathtt{i}_4$ `v`	Dominanz im Intervall $\mathtt{i}_3 \ldots \mathtt{i}_4$	$\{G \mid \exists i\,(\mathtt{i}_3 \leq i \leq \mathtt{i}_4 \wedge v_1 >i\ v_2)\}$
|	`v . v`	direkte Präzedenz	$\{G \mid v_1\ .1\ v_2\}$
|	`v .* v`	Präzedenz	$\{G \mid \exists i\,(v_1\ .i\ v_2)\}$
|	`v >@l v`	linke Ecke	$\{G \mid v_1 > @l\ v_2\}$
|	`v >@r v`	rechte Ecke	$\{G \mid v_1 > @r\ v_2\}$
|	`v .`$\mathtt{i}_3$ `v`	Präzedenz mit der Distanz $\mathtt{i}_3$	$\{G \mid v_1\ .\mathtt{i}_3\ v_2\}$
|	`v .`$\mathtt{i}_3$`,`$\mathtt{i}_4$ `v`	Präzedenz im Intervall $\mathtt{i}_3 \ldots \mathtt{i}_4$	$\{G \mid \exists i\,(\mathtt{i}_3 \leq i \leq \mathtt{i}_4 \wedge v_1.i\ v_2)\}$
|	`v !R v`	negierte Knotenrelation	$\{G \mid !(\,v_1\,\mathtt{R}\,v_2)\}$
|	`v $ v`	Geschwister	$\{G \mid v_1\ \$\ v_2\}$
|	`v $.* v`	Geschwister mit Präzedenz	$\{G \mid v_1\ \$\ v_2 \wedge\ \exists i(v_1\ .i\ v_2)\}$
|	`g & g`	Konjunktion	$[\![\mathtt{g}_1]\!] \cap [\![\mathtt{g}_2]\!]$
|	`g | g`	Disjunktion	$[\![\mathtt{g}_1]\!] \cup [\![\mathtt{g}_2]\!]$

Abbildung 6.1: Syntax und Denotation von Graphbeschreibungen. $v_1 \in [\![\mathtt{v}_1]\!]$; $v_2 \in [\![\mathtt{v}_2]\!]$; $[\![\mathtt{v}_1]\!], [\![\mathtt{v}_2]\!] \subseteq V_G$; `n` Knoten-ID; $\mathtt{i}_1, \mathtt{i}_2 \geq 0$; $\mathtt{i}_3, \mathtt{i}_4 \geq 1$; `R` nicht-negierte Knotenrelation (mit Ausnahme von `$.*`)

6.2.6 Erfüllbarkeit lokaler Beschreibungen

Lokale Beschreibungen, d.h. Attributwerte, Attributspezifikationen, Knoten- und Graphbeschreibungen, werden jeweils als Menge von Strukturen interpretiert, die im Falle von Unterspezifikation aus beliebig vielen Elementen bestehen kann und bei vollständiger Spezifikation einelementig ist.

Die Erfüllbarkeit einer lokalen Beschreibung bezüglich eines Syntaxgraph-Modells $\mathcal{M}$ wird allgemein durch die Denotation der Beschreibung in $\mathcal{M}$ definiert.

Definition 22 (Erfüllbarkeit lokaler Beschreibungen)
Eine Beschreibung γ ($\gamma =$ `d', k, v, g`*) ist erfüllbar in $\mathcal{M} = (C, \mathcal{F}, V, \mathcal{G}, R)$ unter der Variablensubstitution σ ($\mathcal{M}, \sigma \models \gamma$), falls $[\![\gamma]\!]_{\mathcal{M},\sigma} \neq \emptyset$.*

Eine Beschreibung γ ist erfüllbar in $\mathcal{M}$ ($\mathcal{M} \models \gamma$), falls eine Variablensubstitution σ existiert mit $\mathcal{M}, \sigma \models \gamma$.

6.2.7 Korpusdefinition

Eine Korpusdefinition `k` besteht aus einer Attributdeklaration und einer nichtleeren Liste von eingeschränkten Graphbeschreibungen (oder *Korpusgraphen*) $\mathtt{g}^r$. Die Schreibweise g^r zeigt im weiteren Verlauf an, dass es sich um eine eingeschränkte (r für *restricted*) Beschreibung der bezeichneten Struktur handelt.

Zunächst wird die Syntax der eingeschränkten Korpusgraphen festgelegt, anschließend die Attributdeklaration.

Attributwerte

In einer Korpusdefinition dürfen Attributwerte $\mathtt{d'}^r$ weder Typen noch Templates umfassen. Zudem sind boolesche Operatoren ausgeschlossen. Attributwerte sind damit auf Konstanten beschränkt.

$$\begin{array}{lcl} \mathtt{d}^r & \Rightarrow & \mathtt{c} \\ \mathtt{d'}^r & \Rightarrow & \mathtt{d}^r \end{array}$$

Attributspezifikationen

In einer Korpusdefinition dürfen Attributwerte einer Attributspezifikation $\mathtt{k'}^r$ keine regulären Ausdrücke umfassen. Eine Attributspezifikation darf nicht leer sein und als einziger boolescher Operator ist die Konjunktion zugelassen.

$$\begin{array}{lcl} \mathtt{k}^r & \Rightarrow & \mathtt{f = d'}^r \\ & | & \mathtt{k}^r \ \mathtt{\&} \ \mathtt{k}^r \\ \mathtt{k'}^r & \Rightarrow & \mathtt{[\ k}^r \mathtt{\ ]} \end{array}$$

Knotenbeschreibung

In einer Korpusdefinition umfasst jede Knotenbeschreibung $\mathtt{v}^r$ ein Paar bestehend aus einer Knoten-ID und einer nicht-leeren Attributspezifikation.

$$\begin{aligned} \mathtt{n}^r &\Rightarrow \mathtt{c} \\ \mathtt{v}^r &\Rightarrow \mathtt{n}^r\mathtt{:k'}^r \end{aligned}$$

Graphbeschreibungen

In einer Korpusdefinition besteht eine Graphbeschreibung $\mathtt{g}^r$ aus Konjunktionen über den Basisrelationen direkte beschriftete Dominanz und direkte Präzedenz sowie fixierten Stelligkeitsprädikaten.

$$\begin{aligned} \mathtt{g}^r \Rightarrow\ & \mathtt{root(n)} \\ \mid\ & \mathtt{arity(n,a)} \\ \mid\ & \mathtt{v}^r\ \mathtt{>l}\ \mathtt{v}^r \\ \mid\ & \mathtt{v}^r\ \mathtt{.}\ \mathtt{v}^r \\ \mid\ & \mathtt{g}^r\ \mathtt{\&}\ \mathtt{g}^r \end{aligned}$$

Für die Korpusgraphen bestehen ferner folgende Einschränkungen:

1. Die Präzedenzrelation muss auf den Terminalknoten eines Graphen eine totale Ordnung definieren.

2. Für jeden Graphknoten wird seine Stelligkeit explizit definiert.

3. Es gibt genau einen ausgezeichneten Knoten, den sogenannten Wurzelknoten, der niemals als rechte Seite einer beschrifteten gerichteten Dominanzrelation auftritt.

4. Alle Graphknoten ausser dem Wurzelknoten treten genau einmal als rechte Seite einer beschrifteten gerichteten Dominanzrelation auf.

Attributdeklaration

Für jedes Attributsymbol `f` muss es genau eine Attributdeklaration `a` geben (Der Typ `t` ist hier ein Typ auf Attributspezifikations-Ebene und kann die Werte `T`, `NT` und `FREC` annehmen):

Syntax	*Interpretation*
a ⇒ `<feature name="f" domain="t"> <value name="v_1" /> ... <value name="v_n" /> </feature>`	$f : [\![\mathtt{t}]\!] \rightarrow \{v_1, \ldots, v_n\}$ ist eine totale Funktion, d.h. allen Elementen aus $[\![\mathtt{t}]\!]$ wird ein Funktionswert zugeordnet.

Interpretation einer Korpusdefinition

Während eine Konjunktion lokaler Beschreibungen stets als Schnitt ihrer Denotationen interpretiert werden konnte, ist eine Interpretation eines Korpus nur als eine Menge von Mengen von Syntaxgraphen möglich. Um diese Komplizierung zu vermeiden, wird an dieser Stelle für globale Beschreibungen sofort der Erfüllbarkeitsbegriff in einem Modell $\mathcal{M}$ definiert.

Syntax		*Erläuterung*	*Interpretation*
k ⇒	$\mathtt{g}^r$	1-Graph-Korpus	$\mathcal{M} \models \mathtt{g}^r$
|	$\mathtt{g}^r$ k	Korpus	$\mathcal{M} \models \mathtt{g}^r$ und $\mathcal{M} \models$ k

Es sei angemerkt, dass der Skopus einer Variablen stets auf einen einzelnen Syntaxgraphen eingeschränkt ist.

6.2.8 Korpusanfragen

Eine *Korpusanfrage* ist eine Graphbeschreibung.

6.2.9 Typhierarchien

Eine Typhierarchie wird definiert als Menge von *Typdefinitionen* a. Ein Subtyp t' ist entweder ein Typ oder eine Konstante. Ein Typ t wird als disjunkte Vereinigung all seiner Subtypen $\mathtt{t}_i$' interpretiert. Die Interpretationen der Subtypen müssen dabei paarweise disjunkt sein.

Ein Typsymbol t darf höchstens einmal definiert werden, d.h. es tritt höchstens einmal auf der linken Seite einer Typdefinition auf. Jedes Typsymbol muss genau einmal auf der rechten Seite einer Typdefinition verwendet werden (Baumeigenschaft von Typhierarchien).

Syntax			*Erläuterung*	*Interpretation*
`t'`	$\Rightarrow$	`c`		
	|	`t`		
`s`	$\Rightarrow$	`t'`	Subtyp	
	|	`t', s`		
`a`	$\Rightarrow$	`t := s ;`	Typdefinition	$[\![\mathtt{t}]\!] = \bigcup_{1 \leq i \leq n} [\![\mathtt{t}_i\mathtt{'}]\!]$ und $[\![\mathtt{t}_i\mathtt{'}]\!] \cap [\![\mathtt{t}_j\mathtt{'}]\!] = \emptyset$, $1 \leq i < j \leq n$

6.2.10 Templatedefinitionen

Die Syntax und Semantik einer Templatedefinition `s` wird folgendermaßen definiert:

Syntax			*Erläuterung*	*Interpretation*
`s`	$\Rightarrow$	`r' <- g;`	Templateklausel	$\exists \sigma$: Falls $\mathcal{M}, \sigma \models \mathtt{g}$, dann $\mathcal{M}, \sigma \models \mathtt{r'}$

D.h. falls der Templaterumpf `g` gilt, so gilt auch der Templatekopf `r'`. Der Skopus einer Variablen ist auf eine Templateklausel beschränkt.

6.2.11 Der semantische Folgerungsbegriff

Definition 23 (TIGER-Datenbasis)
Sei `k` *eine Korpusdefinition,* `S` *eine Menge von Templatedefinitionen und* `A` *eine Menge von Typdefinitionen. Eine TIGER-Datenbasis* `DB` *ist ein Tripel* $\langle$`k,S,A`$\rangle$.

Definition 24 (Folgerungsbegriff)
Eine Anfrage `g` *folgt aus einer TIGER-Datenbasis* $\mathtt{DB} = \langle \mathtt{k,S,A} \rangle$*, wenn für alle Modelle* $\mathcal{M}$ *gilt:*

Falls $\mathcal{M} \models_{\mathtt{A,S}} \mathtt{k}$*, dann* $\mathcal{M} \models \mathtt{g}$.

Wie oben ist auch hier der Skopus von Variablen auf eine Graphbeschreibung beschränkt, d.h. auf einen Korpusgraph oder eine Anfrage.

Kapitel 7

Korpusdefinition mit TIGER-XML

Obwohl eine gemeinsame Korpusbeschreibungssprache zur Korpusdefinition und Korpusanfrage für die Konzeption des Suchwerkzeugs eine Reihe von Vorzügen bietet, wird in der tatsächlichen Implementation ein XML-basiertes Format verwendet, das zum Korpusdefinitionsformat semantisch äquivalent ist. In Abschnitt 7.1 werden zunächst die Überlegungen skizziert, die zu dieser Trennung geführt haben. Abschnitt 7.2 erläutert anschließend das Design des XML-Formats. Die abschließende Diskussion in Abschnitt 7.3 fasst die zentralen Aussagen dieses Kapitels zusammen.

7.1 Warum eine XML-basierte Korpusdefinition?

7.1.1 Problematik

Die Idee der Verschmelzung von Korpusdefinition und Korpusabfrage zu einer gemeinsamen Korpusbeschreibungssprache schafft für die technische Umsetzung eine Reihe von Vorteilen. Ein erster Prototyp des TIGERSearch-Softwarepakets hat dieses Konzept auch entsprechend umgesetzt. Bei der Anwendung durch die Benutzer ergaben sich jedoch technische Probleme, die im Folgenden diskutiert werden. Als Grundlage der Diskussion soll die Definition eines Beispielgraphen dienen (vgl. auch Abb. 7.1):

```
<sentence id="graphid">

"w1":[word="ein" & pos="ART"] &
"w2":[word="Mann" & pos="NN"] &
"w3":[word="läuft" & pos="VVFIN"] &
"n1":[cat="NP"] &
"n2":[cat="S"] &

"w1" . "w2" & "w2" . "w3" &
```

```
("n2" >SB "n1")  &  ("n2" >HD "w3") &
("n1" >NK "w1")  &  ("n1" >NK "w2") &

arity("n1",2) & arity("n2",2) &

root("n2")

</sentence>
```

Die Diskussion der Problematik lässt sich in zwei Teilbereiche aufteilen:

1. Erzeugen von Korpora

   Das Konzept der Korpusbeschreibungssprache sieht vor, dass Korpora ausschließlich in der Korpusdefinitionssprache definiert werden. Dies hat zur Folge, dass Baumbanken zunächst in das Korpusdefinitionsformat konvertiert werden müssen. Die Tatsache, dass dieses Format nicht standardisiert ist, führt zu folgenden Schwierigkeiten bei der Entwicklung von Konvertierungsprogrammen:

   - Markup

     Bei der Trennung von Inhalt und Markup (mit Markupsymbolen wie `[]`, `&`, `=`, `"` usw.) gibt es Konvertierungsprobleme. Liegt beispielsweise als Wortform der doppelte Anführungsstrich vor, so muss dieser in einer Ersatzkodierung (z.B. `\"`) ausgedrückt werden, da es sonst zu Schwierigkeiten beim Parsen des Dokuments kommt (`word="""`). Durch die fehlende Standardisierung des Formats muss jeder Anwender bei der Entwicklung von Konvertierungsprogrammen wieder dieselben Routinen zur Vermeidung dieser Problematik entwickeln.

   - Zeichensatz

     Das Korpusdefinitionsformat ist textbasiert und setzt damit die Verwendung eines vorgegebenen Zeichensatzes voraus. Sonderzeichen aus anderen Zeichensätzen können zwar in einer Unicode-Kodierung ausgedrückt werden (vgl. Abschnitt 8.3), die Umwandlung in diese Kodierung muss aber in jedem Konvertierungsprogramm wieder neu implementiert werden.

   - Validierung

     Das gravierendste Problem besteht in der fehlenden Validierung der erzeugten Dokumente. Die Implementierung eines Konvertierungsprogramms kann nur dann gelingen, wenn die Korrektheit des generierten Korpusfragments immer wieder überprüft werden kann.

Eine solche Prüfung könnte im Rahmen des Korpusimportprogramms des Suchwerkzeugs geschehen. Doch zum einen ist der Anwender dann immer auf die Verfügbarkeit der Software angewiesen. Zum anderen besteht für den Bereich des Korpusimports das Implementationsproblem, für die Validierung möglichst viele Hinweise anzuzeigen, für den Korpusimport hingegen zu Gunsten einer robusten Verarbeitung auch mit leicht abweichenden Eingaben zurechtzukommen.

2. Weiterverwendung der konvertierten Korpora

   Viele Anwender werden sich nur dann die Mühe einer Konvertierung in das Korpusdefinitionsformat machen, wenn sich diese Arbeit auch über die Verwendung des Suchwerkzeugs hinaus lohnt. Nun ist aber das Korpusdefinitionsformat immerhin so komplex, dass ein Parser für dieses Format nur mit erheblichem Aufwand implementiert werden kann. Der in TIGERSearch verwendete Parser (vgl. Abschnitt 8.3) kann zudem nur in Java-Programmen verwendet werden. Die Verwendung der konvertierten Korpora bleibt damit auf die Korpussuche mit der TIGERSearch-Software beschränkt.

Die genannten Punkte schränken die Praxistauglichkeit des Korpusdefinitionskonzepts enorm ein.

7.1.2 Vorzüge einer XML-basierten Korpusdefinition

Eine Lösungsmöglichkeit besteht in der Verwendung eines XML-basierten Korpusdefinitionsformats. Die Meta-Auszeichungssprache XML ist bereits im ersten Teil dieser Arbeit vorgestellt worden. Für eine Einführung in XML sei auf Fitschen und Lezius (2001a, b) verwiesen. Die Verwendung eines XML-basierten Formats löst zunächst die technischen Anwendungsprobleme:

1. Erzeugen von Korpora

   Bei der Konvertierung von Korpora spielt XML seine Stärken als standardisiertes Dokumentenaustauschformat aus. XML hat als W3C Recommendation (vgl. Bray et al. 2000) einen öffentlichen Standardisierungsprozess durchlaufen. Das Format (d.h. die verwendeten Markup-Symbole und die Art der Kodierung von Inhalt) wird damit von einer weltweiten Nutzergemeinde akzeptiert. Es bleibt also nur noch die Festlegung, wie eine Korpusdefinition strukturiert werden soll.

   Zum Erzeugen von XML-Dokumenten stehen dabei diverse Programmierbibliotheken zur Verfügung, die die Arbeit erleichtern. Stellvertretend sei hier die Open-Source-Bibliothek JDOM für Java genannt (vgl.

http://www.jdom.org). Diese Bibliotheken erlauben den Neuaufbau eines XML-Dokuments als Java-Objektbaum, der anschließend automatisch als XML-Textdokument auf die Festplatte geschrieben wird. Konflikte mit Markupsymbolen (z.B. <>") werden dabei automatisch aufgelöst. Der bei der Erzeugung des Dokuments verwendete Zeichensatz ist frei wählbar.

Die Validierung ist sehr leicht gegen eine bestehende DTD oder gegen ein XML Schema möglich. Ein dazu benötigter XML-Parser steht mittlerweile auf allen Plattformen zur Verfügung. Eine DTD bzw. ein XML Schema stellt daneben auch eine umfassende technische Dokumentation des Korpusdefinitionsformats dar.

2. Weiterverwendung der konvertierten Korpora

   Durch die Verfügbarkeit von XML-Parser-Bibliotheken für alle gängigen Programmiersprachen können die erzeugten Korpora problemlos auch für andere Zwecke verwendet werden. Beispiele sind Konvertierungen in andere Formate über XSLT-Stylesheets oder das Durchsuchen der XML-Korpora mit XML-Suchwerkzeugen (vgl. Abschnitt 3.7).

Es sei an dieser Stelle darauf hingewiesen, dass als Argument gegen die Verwendung von XML immer wieder die speicherintensive Repräsentation der Daten als eine Folge des hohen Gehalts an Meta-Information genannt wird. Im Vergleich zum Korpusdefinitionsformat der Korpusbeschreibungssprache fällt der Mehrbedarf an Speicherplatz mit ca. 25% jedoch maßvoll aus.

Neben den rein technischen Vereinfachungen bringt die Verwendung eines XML-basierten Korpusdefinitionsformats auch konzeptionelle Vorteile:

- Kodierung von Anfrageergebnissen

  XML ist ein leicht erweiterbares Format. Durch Referenzmechanismen können Beziehungen zwischen Inhalten zu bestehenden Kodierungen hinzugefügt werden. Während das ursprüngliche Korpusdefinitionsformat nur für den Zweck der Korpusdefinition eingesetzt werden kann, lässt sich das XML-basierte Format so erweitern, dass auch Anfrageergebnisse (d.h. Angabe des matchenden Subgraphs und der am Match beteiligten Knoten) ausgedrückt werden können. Diese Erweiterung wird in Unterabschnitt 7.2.3 beschrieben.

- Formatkonvertierungen mit XSLT

  Die Verarbeitung von XML-Dokumenten mit XSLT-Stylesheets wird oft als Grund genannt, warum sich XML als Beschreibungsformat durchsetzen konnte (vgl. auch Fitschen und Lezius 2001b). In unserem Anwendungsfeld können XSLT-Stylesheets dazu genutzt werden, die im XML-Korpusdefinitionsformat kodierten Korpora auf standardisierte Art und

Weise in weitere Formate zu konvertieren. Da XSLT-Prozessoren für alle Plattformen verfügbar sind, können solche Stylesheets auch anderen Benutzern zur Verfügung gestellt werden. Die Integration dieser Idee in die TIGERSearch-Architektur ist in Abschnitt 8.2 beschrieben.

Die Fülle der technischen Vereinfachungen und zusätzlichen konzeptionellen Möglichkeiten waren die Hauptgründe für die Entscheidung, das Konzept der Korpusbeschreibungssprache zumindest teilweise aufzuweichen und für die Korpusdefinition ein äquivalentes Format auf der Basis von XML zu entwickeln. Das Design dieses Formats wird im folgenden Abschnitt vorgestellt.

7.2 Design des XML-Formats

Die Anforderungen an das Design des XML-Formats sind klar definiert: Es geht um eine zum ursprünglichen Korpusdefinitionsformat semantisch äquivalente Korpusdefinition, die die Kodierung von Syntaxgraphen erlaubt. Darüber hinaus soll auch die Kodierung von Anfrageergebnissen möglich sein. Der vorliegende Abschnitt beschreibt das entwickelte Format und geht auf die Motivationen ein, die zur Festlegung dieses Formats geführt haben. Die grundlegenden Ideen werden in Mengel und Lezius (2000) beschrieben.[1] Eine technische Beschreibung des TIGER-XML-Formats ist in Lezius et al. (2002a) zu finden.

7.2.1 Der Dokumentheader

Die Korpusdefinition setzt sich aus der Attributdeklaration und der Definition der Korpusgraphen zusammen. Die Attributdeklaration ist bereits XML-konform definiert worden und kann damit ohne jegliche Modifikation übernommen werden (vgl. Beispiel unten).

Um der TIGERSearch-Software und weiteren das TIGER-XML-Format verarbeitenden Programmen auch allgemeine Information über ein vorliegendes Korpus an die Hand zu geben, wird der Attributdeklaration noch Meta-Information beigefügt. Diese wird in einem separaten Element `<meta>` abgelegt und besteht aus folgenden Angaben: Korpusname, Korpusautor, Datum, Korpusbeschreibung und ursprüngliches Korpusformat (z.B. Negra-Format oder Klammerstrukturformat).

Zur Unterscheidung von anderen Korpora wird im Wurzelelement `<corpus>` zusätzlich eine Korpus-ID angegeben. Die Korpusdefinition als Gesamtheit ist nun im Element `<head>` abgelegt und teilt sich in die Angabe von Metainformationen (Element `<meta>`) und die Attributdeklaration (Element `<annotation>`).

[1] Man beachte, dass in diesem Papier eine Vorläuferversion des TIGER-XML-Formats behandelt wird. Daher gibt es geringfügige Abweichungen im Dokumentdesign. Die prinzipiellen Ideen sind jedoch identisch.

Der Dokumentheader für ein Korpus, das nur aus dem Beispielsatz *Ein Mann läuft* (vgl. Abb. 7.1) besteht, sieht nun folgendermaßen aus:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<corpus id="DEMO">

<head>

  <meta>
    <name>Demokorpus</name>
    <author>Wolfgang Lezius</author>
    <format>TIGER-XML-Format</format>
  </meta>

  <annotation>

    <feature name="word" domain="T" />

    <feature name="pos" domain="T">
      <value name="ART">Artikel</value>
      <value name="NN">normales Nomen</value>
      <value name="VVFIN">finites Verb</value>
    </feature>

    <feature name="cat" domain="NT">
      <value name="NP">Nominalphrase</value>
      <value name="S">Satz</value>
    </feature>

    <edgelabel>
      <value name="HD">Kopf</value>
      <value name="NK">Teil des NP-Kerns</value>
      <value name="SB">Subjekt</value>
    </edgelabel>

  </annotation>

</head>
```

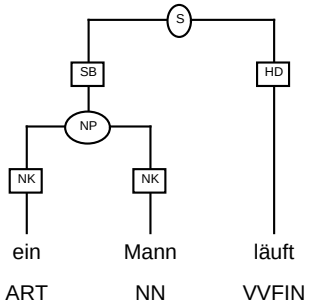


Abbildung 7.1: Beispielsatz *Ein Mann läuft*

7.2.2 Definition eines Syntaxgraphen

Dieser Abschnitt durchläuft die Definition eines Syntaxgraphen von innen nach aussen und zeigt auf, wie die einzelnen Teildefinitionen in XML kodiert werden.

Knoten

Die Knoten und ihre Attributwerte lassen sich in XML sehr intuitiv durch Knotenelemente mit entsprechenden Attributen ausdrücken. So wird beispielsweise aus dem ursprünglichen terminalen Knoten

```
"w1":[word="ein" & pos="ART"]
```

die folgende XML-Darstellung:

```
<t id="w1" word="ein" pos="ART" />
```

Für terminale Knoten werden `<t>`-Elemente vergeben, für nicht-terminale Knoten `<nt>`-Elemente. Die ID rutscht als Attribut in die Elementbeschreibung. Die dadurch entstehende Einschränkung, dass im Korpus kein Attribut mit dem Namen *id* verwendet werden darf, ist als sekundär einzustufen. Eine solche Namensgebung wäre ohnehin missverständlich.

Bei der naheliegenden Modellierung der Korpusattribute als XML-Attribute muss eine besondere Problematik berücksichtigt werden: In einer Dokumenttypdefinition (DTD) muss jedes verwendete Attribut explizit definiert werden. Damit wäre eine DTD aber immer korpusabhängig, da jedes Korpus andere Attribute verwenden darf. Eine Lösung, die oft verwendet wird (z.B. in der XCES-Kodierung, vgl. Abschnitt 2.3), ist die Modellierung der Korpusattribute durch explizite Attribut-Elemente:

```
<t id="w1">
   <attr name="word">ein</attr>
   <attr name="pos">ART</attr>
</t>
```

Bei dieser Lösung können beliebige Korpusattributnamen verwendet werden, eine DTD wäre also korpusunabhängig. Auf der anderen Seite lassen sich in einer DTD Elementinhalte, die aus Zeichenketten und nicht wieder aus Elementen bestehen (PCDATA; hier `ein` und `ART`), nicht weiter einschränken. Beispielsweise wäre eine Auflistung aller Wortarten-Tags zur Sicherung der Konsistenz wünschenswert.

Glücklicherweise steht mit den XML Schemata ein sehr ausdrucksstarker Validierungsmechanismus zur Verfügung, der über die nötigen Mittel verfügt, um zum einen beliebige Attribute zuzulassen und zum anderen auch Zeichenketten-Elementinhalte zu spezifizieren (vgl. `http://www.w3c.org/XML/Schema`). Damit sind also beide Wege technisch durchführbar. XML Schemata sind seit 2001 standardisiert.

Die grundsätzliche Idee beim Design von XML-Dokumenten, dass Attribute ein XML-Elements **beschreiben** sollen, spricht eindeutig für die XML-Attributlösung. Sie macht das XML-Format auch lesbarer. Auch wenn das Format als rein maschinenlesbares Format angesehen wird, so sollte nicht vergessen werden, dass alle Verarbeitungsprogramme (XSLT-Stylesheets, Java-Programme usw.) nach wie vor von Menschen implementiert werden.

Die Validierung durch das XML-Schema wird in Unterabschnitt 7.2.4 beschrieben. Für den Beispielgraphen ergibt sich die folgende Kodierung der drei Terminalknoten:

```
<t id="w1" word="ein"   pos="ART" />
<t id="w2" word="Mann"  pos="NN" />
<t id="w3" word="läuft" pos="VVFIN" />
```

Präzedenz

Die Präzedenz der terminalen Knoten, in der Korpusdefinition ausgedrückt durch eine Konjunktion von Präzedenzrelationen,

```
"w1" . "w2" & "w2" . "w3"
```

braucht in der XML-Kodierung nicht explizit umgesetzt werden. Vielmehr werden alle terminalen Knoten eines Satzes zu einer Gruppe zusammengefasst, d.h. in einem XML-Element `<terminals>` eingebettet. Für den Beispielgraphen sieht die Kodierung der Präzedenz dann folgendermaßen aus:

```
<terminals>
  <t id="w1" word="ein"   pos="ART" />
  <t id="w2" word="Mann"  pos="NN" />
  <t id="w3" word="läuft" pos="VVFIN" />
</terminals>
```

Da die Kinder eines XML-Elements in eindeutiger Weise angeordnet sind, kann die Präzedenz anhand der Position abgelesen werden: Das n-te Kind des `<terminals>`-Elements (also der n-te terminale Knoten) steht hinter dem $(n \Leftrightarrow 1)$-ten und vor dem $(n + 1)$-ten Kind.

Dominanz

Die Realisierung der Dominanzkodierung ist der zentrale Punkt beim Design des XML-Formats. Wenn die ursprüngliche Darstellung des Korpusbeschreibungsformats möglichst erhalten bleiben soll, liegt es eigentlich nahe, die Nichtterminale analog zum vorausgehenden Unterabschnitt aufzulisten und separat eine explizite Liste von Kanten anzugeben. Beispiel:

```
<edge from="n1" to="w1" />
```

Nachteil dieser Vorgehensweise ist, dass ein rekursives Durchwandern des Graphen (z.B. in XSLT-Stylesheets) deutlich schwieriger ist, da eine Kantenkodierung von der Kodierung beider durch die Kante verbundenen Knoten getrennt abgelegt ist.

Eine Alternative zu diesem Ansatz besteht in der Kodierung der Graphstruktur durch Knoteneinbettungen. Kanten werden nicht explizit kodiert, sondern ergeben sich durch die Teil-Ganzes-Beziehung zwischen Mutter- und Tochterknoten. Im Beispiel sähe die Umsetzung der Nominalphrase dann folgendermaßen aus:

```
<nt id="n1" cat="NP">
  <t id="w1" word="ein"   pos="ART" />
  <t id="w2" word="Mann"  pos="NN" />
</nt>
```

Leider trägt auch diese Idee nicht bei der vollständigen Umsetzung des vorliegenden Datenmodells. Zum einen können so keine Kantenbeschriftungen ausgedrückt werden. Für dieses Problem gibt es die Lösung, die Kantenbeschriftung einfach dem nachfolgenden Knoten als zusätzliches Attribut zuzuordnen (z.B. `<t id="w1" edge="NK" />`). Es bleibt aber das Problem, dass diskontinuierliche Konstituenten nicht adäquat beschrieben werden können. Wählt man dennoch die Einbettung zur Kodierung von diskontinuierlichen Konstituenten, so geht die Anordnung der Terminale verloren.

Um die Anordnung der Terminale rekonstruieren zu können, schlagen Bouma und Kloostermann (2002) vor, eine Elementeinbettung vorzunehmen, bei der jedem Knoten Verweise auf seinen linkesten und rechtesten terminalen Nachfolger zugeordnet werden. Auf diese Weise kann eine Dominanzbeziehung direkt abgelesen werden, die Präzedenz zweier Knoten kann hingegen mit Hilfe der linken und rechten Nachfolger geprüft werden. Eine Prüfung der Kontinuität eines Knotens verwendet ebenfalls die Angaben über die linken und rechten Nachfolger. Eine integrierte Kodierung sekundärer Kanten ist bei diesem Ansatz ausgeschlossen.

Der XCES-Ansatz löst den Konflikt dadurch, dass Referenz- und Einbettungsmechanismus als alternative Lösungen angeboten werden, die sogar kombiniert werden können (vgl. Abschnitt 2.3). So bleibt die Anordnung der Terminale zumindest dann explizit, wenn keine diskontinuierlichen Phrasen auftreten. Nachteil dieser Vorgehensweise ist, dass stets beide Wege bei der Weiterverarbeitung unterstützt werden müssen.

Der hier verfolgte Kompromissansatz entscheidet sich für eine Zwischenlösung. Er listet die Nichtterminale analog zu den Terminalen in einer eigenen Gruppe auf und fügt zu jedem Nichtterminalknoten eine Liste aller ausgehenden Kanten an:

```
<nonterminals>
  <nt id="n1" cat="NP">
    <edge idref="w1" />
    <edge idref="w2" />
  </nt>
  <nt id="n2" cat="S">
    <edge idref="n1" />
    <edge idref="w3" />
  </nt>
</nonterminals>
```

Eine Kante besteht nun aus einer Referenz auf den nachfolgenden Knoten. Für das erste `<edge>`-Element bedeutet dies, dass es eine Kante vom Knoten `n1` zum Knoten `w1` gibt.

Kantenbeschriftungen

Die Umsetzung von Kantenbeschriftungen ist damit denkbar einfach. Da Kanten explizit kodiert sind, stellt eine Kantenbeschriftung eine Eigenschaft einer Kante dar, die demnach als Attribut umgesetzt werden sollte. Als Attributname wurde `label` festgelegt. Die Nichtterminale im Beispielgraphen werden damit vollständig auf folgende Weise beschrieben:

```
<nonterminals>
  <nt id="n1" cat="NP">
    <edge label="NK" idref="w1" />
    <edge label="NK" idref="w2" />
  </nt>
  <nt id="n2" cat="S">
    <edge label="SB" idref="n1" />
    <edge label="HD" idref="w3" />
  </nt>
</nonterminals>
```

Die theoretische Konzeption der Korpusbeschreibungssprache geht stets von einem Korpus mit Kantenbeschriftungen aus. Die Implementation unterstützt jedoch auch Korpora, die auf dieses Ausdrucksmittel verzichten. Das TIGER-XML-Format trägt dieser Tatsache Rechnung, indem das `label`-Attribut als optionales Attribut verwendet wird.

Stelligkeit

Es fehlt streng genommen noch die Angabe der Stelligkeit für jeden Nichtterminalknoten. Doch analog zur Anordnung der Terminalknoten ist auch diese Angabe redundant, da sich die Stelligkeit eines Knotens in eindeutiger Weise aus der Anzahl der eingebetteten `<edge>`-Elemente ergibt.

Sekundäre Kanten

Das Konzept der expliziten Kodierung von Kanten lässt auch eine Integration sekundärer Kanten zu. Dazu werden analog zu den `<edge>`-Elementen für primäre Kanten `<secedge>`-Elemente für sekundäre Kanten eingeführt. Ein fiktives Beispiel einer sekundären Kante, die vom `NP`-Knoten zur Wortform `läuft` führt, sähe dann folgendermaßen aus:

```
<nt id="n1" cat="NP">

  <edge label="NK" idref="w1" />
  <edge label="NK" idref="w2" />

  <secedge label="LABEL" idref="w3" />

</nt>
```

Wurzelknoten

Es fehlt noch die explizite Angabe des Wurzelknotens. Dazu wird um die beiden <terminals>- und <nonterminals>-Blöcke ein umrahmendes Element <graph> gespannt, das durch ein Referenz-Attribut auf den Wurzelknoten verweist:

```
<graph root="n2">
  <terminals>
    <t id="w1" word="ein"   pos="ART" />
    <t id="w2" word="Mann"  pos="NN" />
    <t id="w3" word="läuft" pos="VVFIN" />
  </terminals>
  <nonterminals>
    <nt id="n1" cat="NP">
      <edge label="NK" idref="w1" />
      <edge label="NK" idref="w2" />
    </nt>
    <nt id="n2" cat="S">
      <edge label="SB" idref="n1" />
      <edge label="NK" idref="w3" />
    </nt>
  </nonterminals>
</graph>
```

Syntaxgraph

Damit ist die Beschreibung eines Syntaxgraphen fast komplett. Es fehlt die Graph-ID, die in einem weiteren umrahmenden <s>-Element festgelegt wird. Die Bezeichnung <s> steht für einen Satz, da die Korpuseinheiten meist Sätze sind. Das zusätzliche <s>-Element umschließt das <graph>-Element, um auch Anfrageergebnisse aufnehmen zu können (vgl. Unterabschnitt 7.2.3). Der Beispielgraph sieht damit folgendermaßen aus (vgl. Abb. 7.1):

```
<body>

<s id="graphid">

  <graph root="n2">

    <terminals>
      <t id="w1" word="ein"   pos="ART" />
      <t id="w2" word="Mann"  pos="NN" />
      <t id="w3" word="läuft" pos="VVFIN" />
    </terminals>
```

```
    <nonterminals>
      <nt id="n1" cat="NP">
        <edge label="NK" idref="w1" />
        <edge label="NK" idref="w2" />
      </nt>
      <nt id="n2" cat="S">
        <edge label="SB" idref="n1" />
        <edge label="NK" idref="w3" />
      </nt>
    </nonterminals>

  </graph>

</s>

...
</body>
```

Wie im Beispiel bereits angedeutet wird, besteht der Rumpf eines TIGER-XML-Dokuments aus einer Liste von Graphbeschreibungen, die als Geschwisterelemente `<s>` unter einem gemeinsamen `<body>`-Element eingebettet sind. Ein Korpus ist damit vollständig definierbar. Ein ausführliches Korpusbeispiel ist in der technischen Dokumentation zum TIGER-XML-Format abgedruckt (Lezius et al. 2002a).

Zusammenhang zur Korpusdefinition

Die semantische Äquivalenz der Korpusdefinition als Teil der Korpusbeschreibungssprache und der Korpusdefinition mittels des TIGER-XML-Formats ist unmittelbar einleuchtend. Alle Informationen sind in eindeutiger Weise auf das XML-Format abgebildet worden und lassen sich ebenfalls eindeutig aus dem XML-Format wieder ablesen und damit wieder auf die Korpusdefinition abbilden. Die Gegenüberstellung der Graphbeschreibungskodierungen in Abb. 7.2 am Ende dieses Kapitels unterstreicht diese Äquivalenz. Auf einen formalen Beweis wird jedoch an dieser Stelle verzichtet.

7.2.3 Repräsentation von Anfrageergebnissen

Ein großer Vorteil der Verwendung von XML als Basisformat zur Kodierung von Graphen ist die einfache Erweiterbarkeit eines Graphen um Match-Information, d.h. Ergebnisdaten einer Korpusanfrage. Als Beispiel soll die folgende einfache Anfrage dienen, die von einem Teilgraphen des Beispielsatzes erfüllt wird:

```
#n:[cat="NP"] > #w:[pos="NN"]
```

Gesucht wird also nach einer Nominalphrase, die ein Nomen einbettet. Vor der Frage der Kodierung eines Suchergebnisses steht zunächst die Frage, wie ein Suchergebnis überhaupt strukturiert ist.

Ganz allgemein werden diejenigen Korpusgraphen gesucht (repräsentiert durch Graph-IDs), die die Graphbeschreibung der Anfrage enthalten. Doch zur Weiterverarbeitung der Ergebnisse (z.B. grafische Visualisierungen, Häufigkeits- und Kookurrenzanalysen) ist es darüberhinaus wünschenswert zu wissen, welche Knoten an einem Match beteiligt sind (Belegung der Knotenvariablen) und in welchem Subgraphen (repräsentiert durch den Wurzelknoten des Subgraphen) der Match zu finden ist. Der matchende Subgraph ist unter allen Subgraphen, die alle am Match beteiligten Knoten dominieren, der minimale Subgraph (d.h. der selbst keinen der anderen Subgraphen dominiert).

Daneben muss berücksichtigt werden, dass ein Graph durchaus mehrere Matches enthalten kann. D.h. es kann verschiedene Variablenbelegungen eines Graphen geben, die die Anfrage erfüllen (Beispielanfrage: `[cat="NP"]`).

Die durch diese Struktur definierten Anforderungen legen einen Referenzmechanismus nahe, der durch das folgende Beispiel illustriert wird:

```
<s id="graphid">

  <graph>

    <terminals>
      <t id="w1" word="ein"   pos="ART" />
      <t id="w2" word="Mann"  pos="NN" />
      <t id="w3" word="läuft" pos="VVFIN" />
    </terminals>

    <nonterminals>
      <nt id="n1" cat="NP">
        <edge label="NK" idref="w1" />
        <edge label="NK" idref="w2" />
      </nt>
      <nt id="n2" cat="S">
        <edge label="SB" idref="n1" />
        <edge label="NK" idref="w3" />
      </nt>
    </nonterminals>

  </graph>
```

```
  <matches>

    <match subgraph="n1">
      <variable name="#n" idref="n1" />
      <variable name="#w" idref="w1" />
    </match>

  </matches>

</s>
```

Alle Matches innerhalb eines Korpusgraphen werden durch <match>-Elemente innerhalb eines umgebenden <matches>-Elements aufgelistet. Dieses <matches>-Element ist eingebettet in das äußere <s>-Element und ist damit ein Schwesterelement der Graphbeschreibung <graph>.

Ein einzelner Match verweist zunächst auf den Wurzelknoten des matchenden Subgraphen. Diese Information kann streng genommen auch aus den Variablenbelegungen und der Graphstruktur ermittelt werden. Diese Berechnung ist aber insbesondere in XSLT-Stylesheets, die zu den Hauptanwendungen des XML-Formats zählen, nur sehr umständlich zu realisieren.

Innerhalb des Matches werden nun alle Variablen aufgelistet. Der Wert einer Variablen besteht aus einer Referenz auf den Knoten, an den die Variable gebunden ist. Die Variablennamen ergeben sich aus den in der Anfrage verwendeten Variablennamen. Werden keine Namen spezifiziert, so werden in der Implementation zur einheitlichen Weiterverarbeitung der Matchdaten fiktive Ersatznamen generiert. Es empfiehlt sich daher für den Anwender, zugunsten eines lesbaren TIGER-XML-Exports jede Knotenbeschreibung in einer Anfrage mit einem Variablennamen zu versehen. Variablen auf Attributspezifikations-Ebene und Attributwert-Ebene werden nicht aufgelistet, da ihre Belegungen über die angegebene Graphstruktur rekonstruiert werden kann.

Die Kodierung von Anfrageergebnissen ist damit vollständig. Durch den Referenzmechanismus bleibt die Graphdarstellung unangetastet; sie kann also auch völlig isoliert von den Anfrageergebnissen verarbeitet werden.

Welche Möglichkeiten sich durch die konzeptionelle Erweiterung um die Darstellung von Anfrageergebnissen für die Implementation des Anfragewerkzeugs ergeben, wird in Abschnitt 8.2 deutlich.

7.2.4 Validierung des TIGER-XML-Formats

Als Abschluss der Vorstellung des TIGER-XML-Designs sei auf das XML-Schema verwiesen, das zur Validierung von TIGER-XML-Dokumenten verwendet wird. Dieses Schema ist öffentlich zugänglich über die folgende URL:

`http://www.ims.uni-stuttgart.de/projekte/TIGER/public/TigerXML.xsd`

Eine technische Beschreibung des TIGER-XML-Formats, die auch eine Kurzbeschreibung des Schemas enthält, ist in Lezius et al. (2002a) nachzulesen.

Es sei zudem bemerkt, dass dieses allgemeine Schema noch nicht die Ideallösung zur Validierung von TIGER-XML-Dokumenten darstellt, da es für die Terminal- und Nichtterminalknoten beliebige Attributnamen zulässt. Vielmehr sollte jedes einzelne Korpus ein eigenes Schema verwenden, das das gemeinsame Basisschema erweitert und die Korpusattribute (Namen und Wertebereiche) im Rahmen dieser Erweiterung genau spezifiziert. XML Schemata stellen dazu einen Mechanismus zur Verfügung, der sich am Vererbungsprinzip orientiert.

7.2.5 Diskussion des TIGER-XML-Formats

In diesem Abschnitt ist die Konzeption des TIGER-XML-Formate beschrieben worden. Doch wie ist diese Konzeption im Vergleich zu XML-basierten Kodierungsansätzen wie XCES zu bewerten?

Zunächst ist festzuhalten, dass der TIGER-XML-Ansatz keine stand-off Annotation vorsieht. Nachteil dieser Vorgehensweise ist die fehlende bzw. erschwerte Erweiterbarkeit des Formats um andere Annotationsebenen. Auf der anderen Seite ist eine solche Erweiterbarkeit nicht als Anforderung an das zu entwickelnde Suchwerkzeug vorgesehen. Das TIGER-XML-Format stellt eine auf das Import-Problem des TIGERSearch-Werkzeugs zugeschnitte Lösung dar. Daneben ist die Verwaltung aller Korpusdaten innerhalb einer einzigen Datei aus technischer Sicht die einfachste Lösung. Anwendungen wie der Korpusimport oder die Konvertierung in andere Format mit XSLT-Stylesheets können auf genau einem XML-Dokument arbeiten und müssen sich nicht um die Auflösung von Referenzen auf sekundäre Dateien kümmern.

Um die Standardisierungsinitiative XCES zu unterstützen, wäre eine Verwendung des XCES-Vorschlags zur Kodierung syntaktischer Annotation eine Alternative gewesen. Doch ist zum gegenwärtigen Stand noch unklar, inwieweit dieser Vorschlag von der Forschungsgemeinschaft angenommen wird. Bei entsprechend hoher Akzeptanz kann eine Unterstützung in Form eines Importfilters (vgl. Abschnitt 8.1) oder eine direkte Unterstützung als zweites Korpuseingangsformat in Betracht gezogen werden.

7.3 Zusammenfassung

Im Rahmen dieses Kapitels ist die Zielvorstellung, Korpusdefinition und Korpusanfrage in einer gemeinsamen Sprache auszudrücken, ein Stück weit aufgegeben worden. Es bleibt die Frage, ob sich dieser Bruch in der Konzeption auch

wirklich gelohnt hat. Diese Frage ist natürlich nicht ad hoc zu beantworten, sondern muss eigentlich der Anwenderschaft gestellt werden.

Kapitel 8 soll jedoch richtungsweisend aufzeigen, welche Vorteile das veränderte Design für die Implementation und damit für den Endanwender mit sich bringt.

Was die Neuausrichtung der Korpusdefinition anbelangt, so ist mit dem XML-Format kein wirkliches Neuland betreten worden. Die Attributdeklaration konnte ohne Modifikation übernommen werden und die Definition der Graphbeschreibungen zeichnet sich fast ausschließlich durch Umformatierung und weniger durch Umstrukturierungen aus (vgl. auch Abb. 7.2). Das XML-Format stellt damit lediglich eine leicht abgewandelte Darstellung der ursprünglichen Korpusdefinition dar.

```
<sentence id="graphid">

root("n2") &

"w1":[word="ein" & pos="ART"] &
"w2":[word="Mann" & pos="NN"] &
"w3":[word="läuft" & pos="VVFIN"] &

"w1"."w2" & "w2"."w3" &

"n1":[cat="NP"] &
"n1" >NK "w1" &
"n1" >NK "w2" &
arity("n1",2) &

"n2":[cat="S"] &
"n2" >SB "n1" &
"n2" >HD "w3" &
arity("n2",2)

</sentence>
```

```
<s id="graphid">

<graph root="n2">

  <terminals>

    <t id="w1" word="ein" pos="ART" />
    <t id="w2" word="Mann" pos="NN" />
    <t id="w3" word="läuft" pos="VVFIN" />

  </terminals>

  <nonterminals>

    <nt id="n1" cat="NP">
      <edge label="NK" idref="w1" />
      <edge label="NK" idref="w2" />
    </nt>

    <nt id="n2" cat="S">
      <edge label="SB" idref="n1" />
      <edge label="NK" idref="w3" />
    </nt>

  </nonterminals>

</graph>

</s>
```

Figure 7.2: Vergleich der Korpusdefinitionsformate

Kapitel 8

Implementation der Korpusbeschreibungssprache

Nachdem in den vorausgegangenen Kapiteln die Konzeption der Korpusbeschreibungssprache vorgestellt wurde, geht es im vorliegenden Kapitel um deren Implementation. Bedingt durch die teilweise Aufweichung des Korpusbeschreibungssprachen-Konzepts sind auch bei der Implementation mehrere Stränge der Umsetzung zu unterscheiden.

Im ersten Abschnitt 8.1 wird aufgezeigt, wie das TIGER-XML-Format als Korpuseingangsformat eingesetzt wird und dabei als Schnittstelle zwischen dem TIGERSearch-Softwarepaket und beliebigen Korpusformaten fungiert. Das nachfolgende Abschnitt 8.2 widmet sich dem direkten Gegenstück, dem Export von Anfrageergebnissen in Form von TIGER-XML-Dokumenten. Anschließend wird in Abschnitt 8.3 gezeigt, wie Ausdrücke der Anfragesprache geparst werden, um sie mit dem Anfrageprozessor zu verarbeiten. Der abschließende Abschnitt 8.4 fasst die Ergebnisse der Implementation und damit auch diesen Teil der Arbeit zusammen.

8.1 Import von Korpora mit TIGER-XML

Die zentrale Idee des Korpusbeschreibungsformats besteht darin, Korpusdefinition und Korpusanfrage in derselben Sprache auszudrücken. Als Konsequenz werden Korpora in der Teilsprache zur Definition kodiert. Diese Rolle nimmt das TIGER-XML-Format ein (vgl. Abb. 8.1).

An dieser Stelle erfolgt zum besseren Verständnis ein Vorgriff auf den dritten Teil dieser Arbeit, also die Konzeption der Auswertung von Korpusanfragen (vgl. Kapitel 10). Für die Konzeption der Anfragesoftware wird ein zweigeteiltes, indexbasiertes Konzept verfolgt. Um die Verarbeitung von Suchanfragen zu beschleunigen, wird ein neues Korpus stets einem Vorverarbeitungsschritt, der sogenannten Indexierung, unterzogen. In diesem Vorgang werden kosteninten-

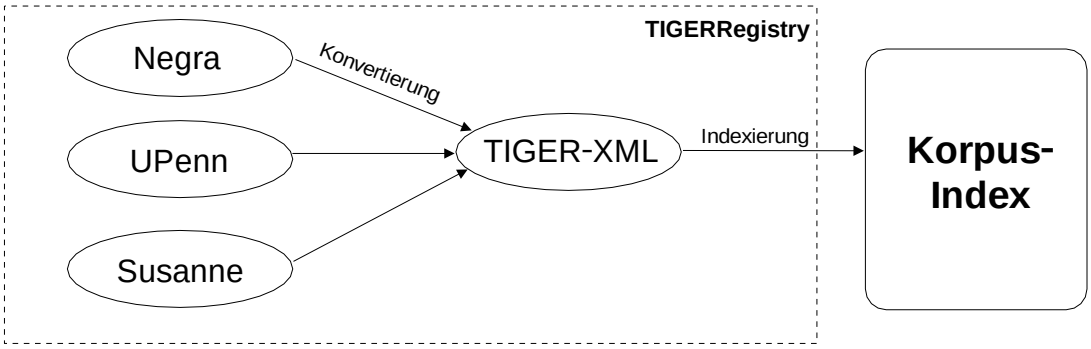


Abbildung 8.1: Architektur des Korpusindexierungs-Werkzeugs

sive partielle Suchvorgänge durchgeführt und die Ergebnisse der Suchvorgänge, der sogenannte Index, in binärer (d.h. nicht text-basierter) Form auf der Festplatte abgelegt.

Die Indexierung wird in der Anfragesoftware durch ein eigenständiges Korpusverwaltungsprogramm realisiert, das *TIGERRegistry* genannt wird (vgl. Abb. 8.1 bzw. Abschnitt 14.2). Die einzelnen Bestandteile des Index werden in Kapitel 11 beschrieben. Das ebenfalls eigenständige Anfrageverarbeitungswerkzeug *TIGERSearch* liest die Indexdaten eines Korpus ein und führt anhand dieser Daten die Auswertung von Anfragen durch (vgl. auch Abb. 8.2 bzw. Abschnitt 14.3).

Doch zurück zum Importformat TIGER-XML. Wie Abb. 8.1 zeigt, ist hier eine zweischrittige Korpusaufbereitung vorgesehen. Das TIGER-XML-Format stellt die alleinige Schnittstelle zur Indexierung dar. Korpora in anderen Formaten wie die PennTreebank oder das Susanne Korpus müssen zunächst in das TIGER-XML-Format konvertiert werden.

Bei dieser Konvertierung handelt es sich **nicht** um eine reine Format-Transformation. Hier sind zusätzlich linguistische Reinterpretationen notwendig. Man bedenke etwa, dass Spuren, die in der PennTreebank u.a. zur Repräsentation von Bewegungsphänomenen verwendet werden (vgl. Marcus et al. 1993, 1994), im vorliegenden Datenmodell durch ein eigenes `trace`-Attribut oder auch durch sekundäre Kanten modelliert werden können. Die tatsächliche Modellierung ist das Ergebnis eines linguistischen Entscheidungsprozesses, dessen Ergebnis nie unstrittig sein kann.

Die Konvertierungsprogramme werden auch als Importfilter bezeichnet, da sie Korpora in anderen Formaten einlesen, die im TIGER-XML-Format verarbeitbare Information herausfiltern und ggf. umstrukturieren und als Ergebnis eine TIGER-XML-Variante des Originalkorpus generieren. Damit nicht jeder Anwender seine eigenen Importfilter entwickeln muss, werden einige gängige durch das TIGER-XML-Format modellierbare Korpusformate durch bereits implementierte Importfilter unterstützt. Zu den unterstützten Korpusformaten zählen u.a.

das Klammerstrukturformat, das PennTreebank-Format als erweitertes Klammerstrukturformar mit Funktionen und Spuren, das Susanne- und Christine-Format sowie das Negra-Format. Eine Liste aller implementierten Korpusfilter ist im TIGERRegistry-Manual zu finden (vgl. Lezius et al. 2002b).

Die nächste konzeptionelle Stufe des Importfilterkonzepts ist ein PlugIn-Mechanismus. Für die Anwender der Sofware steht eine Programmierschnittstellenbeschreibung für Importfilter zur Verfügung. Es handelt sich hier um eine abstrakte Java-Klasse, die von jedem Importfilter implementiert werden muss. Hält sich der Entwickler des Importfilters an diese Schnittstelle, so kann er seinen Filter vollständig in die grafische Oberfläche des TIGERRegistry-Tools integrieren.

8.2 Export von Anfragergebnissen mit TIGER-XML

8.2.1 Der TIGER-XML-Export

Die größten Vorzüge der Verwendung des TIGER-XML-Formats treten beim Export von Anfrageergebnissen zu Tage. Das Konzept sieht dabei folgendermaßen aus (vgl. Abb. 8.2): Eine Anfrage des Benutzers wird zunächst geparst (vgl. Abschnitt 8.3) und dann vom Anfrageprozessor verarbeitet. Der Prozessor wird hier als *black box* betrachtet, deren genaue Funktionsweise im dritten Teil dieser Arbeit beschrieben wird. Die Ergebnisse der Anfrage liegen dann als Java-Objekt vor.

Es besteht nun zum einen die Möglichkeit, sich die Ergebnisse im sogenannten GraphViewer anzuschauen. Es handelt sich hierbei um ein Visualisierungswerkzeug für Korpusgraphen, das in Abschnitt 14.4 vorgestellt wird. Zum anderen können die Anfrageergebnisse als TIGER-XML-Datei exportiert werden. Dabei können wahlweise alle matchenden Graphen, eine Auswahl der matchenden Graphen, alle nicht-matchenden Graphen, oder auch das ganze Korpus ausgewählt werden.

Doch welche Möglichkeiten der Weiterverarbeitung bietet nun eine solche TIGER-XML-Darstellung der Anfrageergebnisse? Die folgende Auflistung zeigt die drei häufigsten Anwendungen auf und stellt als Abschluss vor, wie die XSLT-Stylesheet-Anwendung direkt in das Anfragewerkzeug integriert worden ist.

Verwendung als XML-Dokument

Ein exportiertes TIGER-XML-Korpus stellt ein eigenständiges XML-Dokument dar. Es kann damit mit allen verfügbaren XML-Werkzeugen weiterverarbeitet werden. Zu den für ein Korpus interessanten Anwendungen zählt dabei die Suche auf dem XML-Korpus mit Hilfe eines XML-Suchwerkzeugs (vgl. Kapitel 3.7).

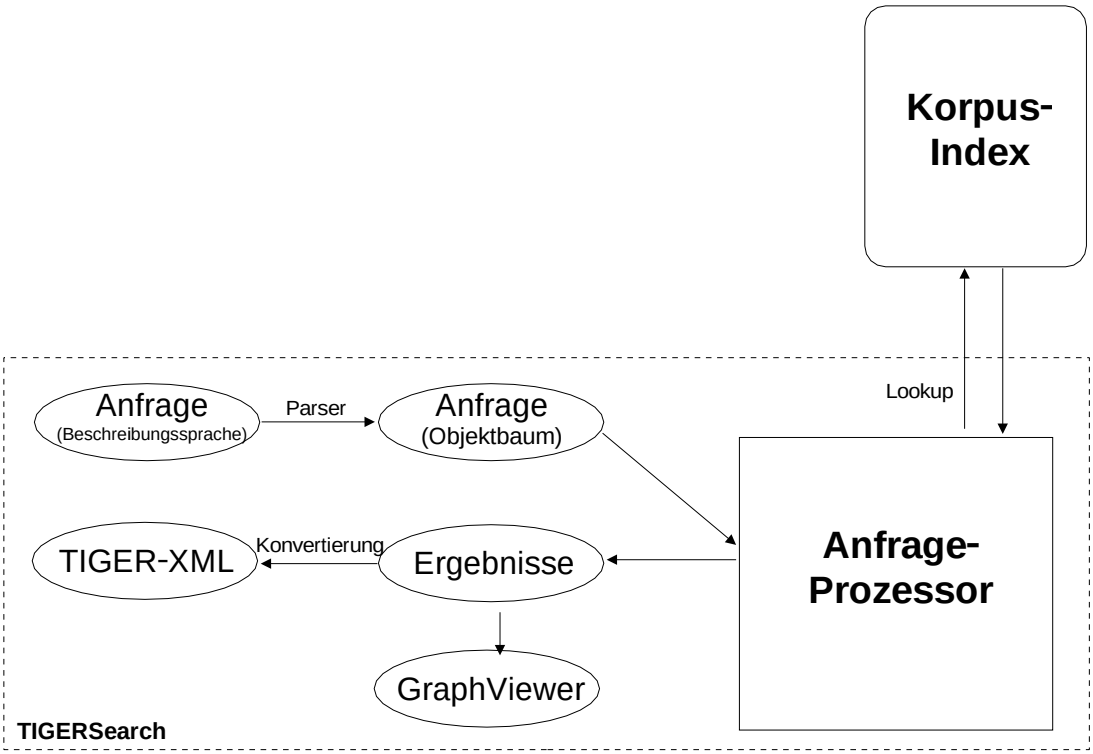


Abbildung 8.2: Architektur des Korpusanfrage-Werkzeugs

Verwendung als neues TIGERSearch-Korpus

Eine offensichtliche Weiterverwendung eines exportierten TIGER-XML-Korpus besteht in der Neuindexierung durch das TIGERRegistry-Werkzeug. Aus einem relativ umfangreichen Korpus kann so durch eine gezielte Anfrage ein Teilkorpus gewonnen werden, mit dem effizienter weitergearbeitet werden kann.

Falls das neu entstandene Korpus jedoch nur für einige verfeinernde Anfragen genutzt werden soll, ist dieser Vorgang zu aufwändig. Bei umfangreichen linguistischen Untersuchungen lohnt sich dagegen der Mehraufwand der Neuindexierung, da das neu entstandene Korpus auch über mehrere TIGERSearch-Sitzungen hinweg als eigenständiges Korpus genutzt werden kann.

Formatkonvertierung durch XSLT-Stylesheets

Die *Extensible Stylesheet Language for Transformation* XSLT ist zur Transformation von XML-Dokumenten in andere Formate entwickelt worden (vgl. Clark 1999). Wie das XSLT-Verarbeitungsmodell zeigt (vgl. Abb. 8.3), bildet das XML-Dokument die Grundlage des Transformationsprozesses, für den ein so genannter Stylesheet-Prozessor benötigt wird. Er wendet die Transformationsregeln an, die in den XSLT-Stylesheets beschrieben werden und legt die Ausgabe in der

Zieldatei ab. XSLT-Stylesheets sind selbst wieder XML-Dateien, womit zur Bearbeitung von Stylesheets alle verfügbaren XML-Tools genutzt werden können. Mittlerweile stehen Stylesheet-Prozessoren für alle Plattformen zur Verfügung. Als Zielformate kommen beliebige textbasierte Formate wie XML, HTML, WML oder auch Textverarbeitungsformate wie LaTeX in Frage. Für ein Einführungsbeispiel in die Verwendung von XSLT-Stylesheets sei auf Fitschen und Lezius (2001b) verwiesen.

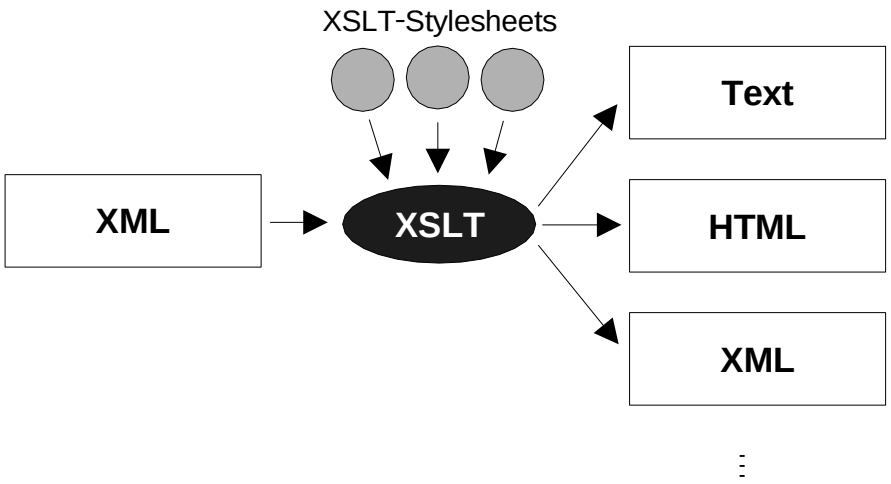


Abbildung 8.3: Das XSLT-Verarbeitungsmodell (aus: Fitschen und Lezius 2001b)

Für die Weiterverarbeitung der TIGER-XML-Dokumente sind vor allem Formatkonvertierungen oder Matchvisualisierungen interessant. So können die exportierten Sätze beispielsweise in eine Klammerstruktur-Darstellung überführt werden. Für die Matches ist etwa eine HTML- oder LaTeX-Darstellung aller matchenden Sätze mit Hervorhebung der matchenden Token durch Farb- bzw. Fettdruck hilfreich. Einige Beispiel-Stylesheets werden mit der TIGERSearch-Software ausgeliefert. Sie sind im TIGERSearch Manual beschrieben (vgl. Lezius et al. 2002c).

Das folgende Beispiel-Stylesheet zeigt, mit welch geringem Aufwand sich die TIGER-XML-Darstellung verarbeiten lässt. Dieses Stylesheet ignoriert die Match-Information und gibt stattdessen den Satz Token für Token aus. Einem XSLT-Stylesheet liegt eine Sammlung so genannter Templates zugrunde. Sie beschreiben, wie Bestandteile des XML-Dokumentbaums (Element, Attribute usw.) transformiert werden. Gestartet wird der Transformationsprozess mit dem Wurzelelement des XML-Dokuments. Die Beschreibung des Transformationsprozesses durch sich gegenseitig aufrufende Templates orientiert sich an der funktionalen Programmierung (Prolog, Lisp usw.). Deshalb finden sich hier diejenigen Anwender besonders gut zurecht, die aus diesem Umfeld kommen.

```
<xsl:stylesheet>

 <xsl:output method="text" encoding="ISO-8859-1" />

 <xsl:template match="corpus">
   <xsl:apply-templates select="body/s" />
 </xsl:template>

 <xsl:template match="s" >
   <xsl:apply-templates select="graph/terminals/t" />
   <xsl:text>&#10;&#10;</xsl:text> <!-- Leerzeile nach Satzende -->
 </xsl:template>

 <xsl:template match="t" >
   <xsl:value-of select="@word"/>
   <xsl:text> </xsl:text> <!-- Leerzeichen nach jedem Token -->
 </xsl:template>

</xsl:stylesheet>
```

Ein XSLT-PlugIn-Konzept

Analog zum Import ist auch für den Export ein PlugIn-Konzept realisiert worden. In diesem Fall werden jedoch keine Java-Implementationen, sondern XSLT-Stylesheets an die TIGERSearch-Software gebunden. Alle in einem festgelegten Verzeichnis befindlichen Stylesheets werden auf der Benutzeroberfläche als zusätzliche Alternativen zum reinen XML-Export angeboten.

Wird ein Stylesheet ausgewählt, so werden die selektierten Korpusgraphen mit Hilfe eines in Java implementierten Stylesheet-Prozessors durch das Stylesheet geschleust und das Ergebnis der Stylesheetprozessierung in die Ergebnisdatei geschrieben. Das TIGER-XML-Dokument wird also nur zur Laufzeit virtuell generiert und nach der Prozessierung aus dem Hauptspeicher entfernt.

Es sei noch bemerkt, dass aus Speicherplatzgründen nicht das gesamte TIGER-XML-Dokument am Stück erzeugt und verarbeitet wird, sondern eine graphenweise Verarbeitung stattfindet. Für n Korpusgraphen werden also n TIGER-XML-Dokumente mit je einem Graphen erzeugt und n Stylesheet-Prozessierungen durchgeführt. Die Exportlaufzeit wird zwar bedingt durch den etwas trägen Initialisierungsprozess des Stylesheet-Prozessors ein wenig gebremst. Auf der anderen Seite bleibt der Hauptspeicherbedarf für den Export so konstant und hängt nicht von der Anzahl der selektierten Graphen ab.

Wie sich dieses Verfahren auf der Benutzeroberfläche darstellt, kann im TIGERSearch-Manual (vgl. Lezius et al. 2002c) nachgelesen werden.

8.2.2 Eine API-Schnittstelle für den Graphviewer

Neben dem Import von Korpora und dem Export von Anfrageergebnissen hat sich für das TIGER-XML-Format noch ein weiteres Anwendungsfeld eröffnet. Dabei geht es um den GraphViewer, der zur Visualisierung von Anfrageergebnissen eingesetzt wird (vgl. Abb. 8.3). Um dieses Visualisierungswerkzeug auch in anderen Programmen einsetzen zu können, werden sowohl eine Programmierschnittstelle als auch eine Datenschnittstelle benötigt.

Für die Datenschnittstelle ist das TIGER-XML-Format ideal geeignet. Denn hier kommen wieder alle Vorzüge eines XML-basierten Austauschformats (Standardisierung, austauschbarer Zeichensatz usw.) voll zum Tragen. Als zusätzlicher Vorteil kann festgehalten werden, dass der GraphViewer mit einer TIGER-XML-Schnittstelle, die auch Anfrageergebnisse kodieren kann, von anderen Programmen nicht nur zur reinen Visualisierung von Syntaxgraphen verwendet werden kann. Daneben stehen auch alle Match-Visualisierungsmechanismen zur Verfügung, die etwa zur Fokussierung von Teilstrukturen eingesetzt werden können.

8.3 Implementation als Korpusanfragesprache

Als verbleibendes Einsatzgebiet der Korpusbeschreibungssprache ist noch die Verwendung als Korpusanfragesprache zu nennen. Aufgabe des Anfrage-Parsers ist es, Ausdrücke in dieser Sprache auf syntaktische Korrektheit zu prüfen und die syntaktische Struktur als Java-Objekt-Baum zu repräsentieren (vgl. Abb. 8.3).

Als Hilfsmittel ist die Java-Bibliothek *JavaCC* eingesetzt worden (vgl. `http://www.webgain.com/products/java_cc/`). Sie erhält eine attributierte Grammatik der Anfragesprache als Eingabe. Auf dieser Basis werden Java-Klassen erzeugt, die das Parsen der syntaktischen Struktur und die Erzeugung des die Struktur repräsentierenden Java-Objekt-Baums übernehmen.

Um einen Eindruck davon zu bekommen, wie das Ergebnis des Parsing-Prozesses aussieht, ist als Beispiel der Java-Objekt-Baum für eine einfache Anfrage abgedruckt (Abb. 8.4). Die Attribute der Objekte (z.B. das genaue Relationssymbol der im Beispiel verwendeten Relation) sind jeweils in Klammern angedeutet.

```
#np:[cat="NP"] > #noun:[pos="NN"]
```

8.4 Zusammenfassung

Abb. 8.5 zeigt den erreichten Zwischenstand der Konzeption und Implementation. Durch das TIGER-XML-Format konnten sowohl das Korpuseingangsformat

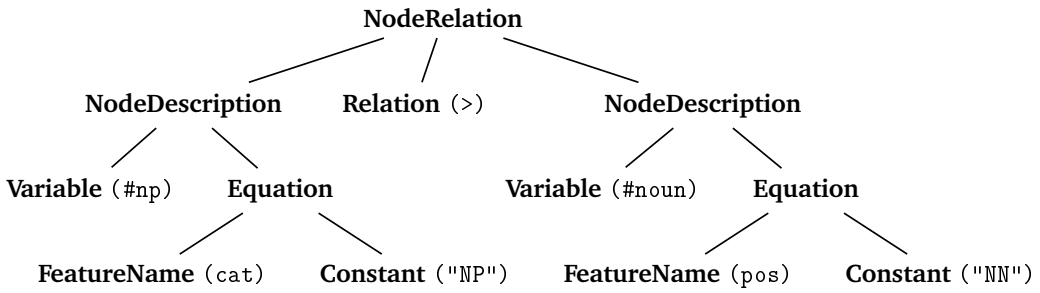


Figure 8.4: Java-Objekt-Baum als Parse-Ergebnis einer Anfrage

wie auch die Kodierung von Anfrageergebnissen abgedeckt werden. Die allgemeine Korpusbeschreibungssprache wird als Anfragesprache verwendet. Es gilt nun, mit der Verarbeitung der Anfragen den zentralen Baustein der Architektur zu realisieren.

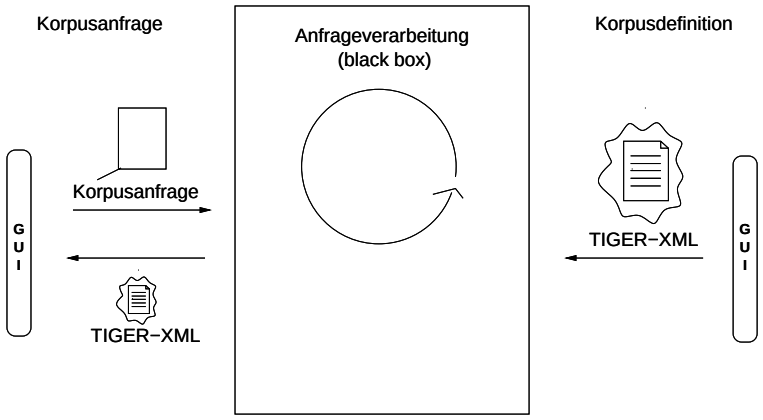


Abbildung 8.5: Konzeption des Werkzeugs - Zwischenstand

Teil III

Verarbeitung von Korpusanfragen

Kapitel 9

Der Kalkül

Im vorangegangenen Teil dieser Arbeit ist eine Sprache zur Definition und Abfrage eines Korpus konzipiert und formalisiert worden. Der vorliegende Teil der Arbeit behandelt die Implementation der Anfrageverarbeitung. Die Implementation hat die Aufgabe, möglichst effizient zu prüfen, ob eine Anfrage aus einer gegebenen Datenbasis (d.h. Korpus + Typdefinitionen + Templates) folgt.

Eine prinzipiell denkbare Vorgehensweise besteht darin, die denotationelle Semantik der Beschreibungssprache als Verarbeitungsgrundlage zu verwenden. Da der Kern einer solchen Implementation jedoch aus dem Raten von Variablenbelegungen bestehen würde, ist diese Vorgehensweise unpraktikabel.

Stattdessen wird mit dem so genannten TIGER-Kalkül ein Bindeglied zwischen denotationeller Semantik und einer effizienten Implementation geschaffen (vgl. Abb. 9.1). Dieser Kalkül definiert einen syntaktischen Ableitungsbegriff. Das Raten von Variablenbelegungen wird dabei durch ein Constraint-Löse-Verfahren ersetzt.

Das vorliegende Kapitel definiert den TIGER-Kalkül. Dieses Kapitel stellt eine Zusammenfassung eines Arbeitsberichts dar, der den Kalkül ausführlich beschreibt und auch auf seine Korrektheit und Vollständigkeit eingeht (König und Lezius 2002b).

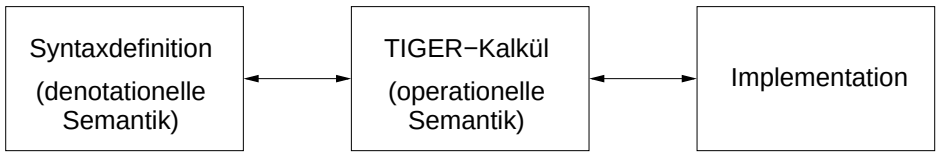


Abbildung 9.1: Der TIGER-Kalkül als Bindeglied

Die Verwendung des Verarbeitungskalküls als Bindeglied zwischen denotationeller Semantik und Implementation (vgl. Abb. 9.1) hat folgende Vorteile:

- Die Definition der Sprachsyntax und die Definition des Kalküls stellen eine konzeptuelle Spezifikation einer Implementation dar. Sie strukturieren eine mögliche Implementation vor, ohne die genaue Umsetzung vorwegzunehmen.

- Der Nachweis der Korrektheit und Vollständigkeit der Implementation wird zweischrittig geführt und ist deshalb weniger komplex. Zum einen muss die Korrektheit und Vollständigkeit des TIGER-Kalküls bzgl. der denotationellen Semantik nachgewiesen werden. Zum anderen muss gezeigt werden, dass die Implementation den Kalkül korrekt und vollständig realisiert.

- Die Definition des Kalküls stellt eine wichtige Dokumentation der Implementation dar. Während der Java-Quellcode die Realisierung im Detail widerspiegelt (verwendete Datenstrukturen usw.), dokumentiert die Kalküldefinition die generelle Vorgehensweise.

Der TIGER-Kalkül enthält zwei nicht-deterministische Regeln: Raten eines matchenden Korpusgraphen und Raten eines matchenden Korpusknotens. Eine Implementation kann den Nichtdeterminismus zwar durch eine erschöpfende Suche auflösen, für eine effiziente Verarbeitung sollten diese beiden Punkte jedoch auf möglichst geschickte Art und Weise umgesetzt werden. Im Kalkül werden zudem Disjunktionen (mit Ausnahme der Attributwert-Ebene) zur Vereinfachung der Konzeption auf die äußerste Formel-Ebene ausmultipliziert, d.h. eine Disjunktive Normalform erzeugt. Aus diesem Grund sollte eine Implementation effizient mit Disjunktionen umgehen können.

Es sei an dieser Stelle angemerkt, dass die Korpusbeschreibungssprache zur Vereinfachung weiter eingeschränkt worden ist: Variablen dürfen nicht im Skopus von Negation stehen. Der Begründungszusammenhang, der zu dieser Entscheidung geführt hat, wird in König und Lezius (2002b) ausführlich dargelegt.

9.1 Einführung

Der TIGER-Kalkül hat die Aufgabe, die in der formalen Semantik definierte Folgerung einer Anfrage aus einer TIGER-Datenbasis (vgl. Kapitel 6) syntaktisch zu überprüfen. Das eingesetzte Verarbeitungsmodell orientiert sich dabei konzeptionell am Resolutionskalkül für Horn-Klauseln in logischen Programmiersprachen.

Jeder im Korpus definierte Syntaxgraph entspricht bei dieser Sichtweise einem Fakt in der Datenbasis. Ein Syntaxgraph kann dabei als Argument eines Prädikats `graph` aufgefasst werden. Eine Anfrage wird dann analog mit einem

`graph`-Prädikat umschlossen. Anfragen werden durch eine angepasste Resolutionsregel verarbeitet: Statt Termen erster Ordnung werden hier Graphbeschreibungen (mit eingebetteten Attributspezifikationen, die wieder Attribut-Wert-Paare einbetten) betrachtet. Termunifikation wird zudem durch ein Constraint-Löse-Verfahren (Attributspezifikationsprüfung usw.) ersetzt.

Als Datenbasis wird eine Tiger-Datenbasis vorausgesetzt, die aus der Korpusdefinition, einer Menge von Typdefinitionen und einer Attributdeklaration besteht. Die folgende Typdefinition ist zudem Bestandteil jedes Korpus:

$$\top \texttt{ := Constant, FRec, Node, Graph;}$$

Statt Variablensubstitutionen zu betrachten, werden im Kalkül eine Constraint-Halde Σ und eine Variablenumbenennungs-Halde Θ verwendet. Diese Sichtweise ist deutlich näher an einer späteren Implementation.

Unter einer Constraint-Halde wird eine Menge von Paaren $\langle \texttt{\#x}, \phi \rangle$ verstanden, in der einer Variablen eine Bedingung (*Constraint*) zugeordnet wird. Eine Bedingung kann dabei ein Attribut-Wert `d`, eine Attribut-Spezifikation `k`, oder ein Paar $\langle \texttt{c}, \texttt{\#x} \rangle$ bestehend aus einer Knoten-ID (oder $\top$-Symbol) und einer Attributspezifikationsvariablen sein. Die Constraint-Halde ist zu Beginn der Anfrageverarbeitung leer. Damit der Lookup einer Variablen dennoch wohldefiniert ist, wird für eine nicht in der Halde auftretende Variable `#x` die Existenz eines Paares $\langle \texttt{\#x}, \top \rangle$ bzw. $\langle \texttt{\#x}, \langle \top, \texttt{\#x'} \rangle \rangle$ (`#x'` ungebundene Attributspezifikations-Variable) angenommen.

Die Variablenumbenennungs-Halde Θ, die aus Paaren von Umbenennungen $\langle \texttt{\#x}_1, \texttt{\#x}_2 \rangle$ besteht, ist zu Beginn ebenfalls leer. Zur Verwendung der Halde wird die Existenz einer Abbildung `deref(#x)` angenommen, die miteinander unifizierte Variablen stets auf die in der Formel verbliebene Variable zurückführt. Sie ist folgendermaßen definiert:

$$\texttt{deref(\#x}_1\texttt{)} := \begin{cases} \texttt{\#x}_1 & \text{falls } \langle \texttt{\#x}_1, \texttt{\#x}_2 \rangle \notin \Theta \\ \texttt{deref(\#x}_2\texttt{)} & \text{falls } \langle \texttt{\#x}_1, \texttt{\#x}_2 \rangle \in \Theta \end{cases}$$

Die folgenden vier Abschnitte 9.2 bis 9.5 definieren die vier Verarbeitungsstufen des Kalküls. Die Unterabschnitte sind in Verarbeitungsblöcke gegliedert, die solche Verarbeitungsregeln zusammenfassen, die einem gemeinsamen Zweck dienen. Eine Dokumentation jedes Verarbeitungsblocks soll das Verständnis des Kalküls erleichtern. Alle vier Verarbeitungsblöcke gehen von der Verarbeitung genau eines Korpusgraphen aus. Abschnitt 9.6 verallgemeinert die Verarbeitung und definiert die Ableitbarkeit aus einem Korpus. Abschließend illustriert Abschnitt 9.7 die Konzeption des Kalküls an einem Verarbeitungsbeispiel.

9.2 Verarbeitung von Attributwerten

9.2.1 Disjunktive Normalform

Die Disjunktive Normalform führt auf Attribut-Wert-Ebene zu einer Disjunktion von Konjunkten, die aus Literalen (Konstanten, negierten Konstanten, Typen und negierten Typen) bestehen.

Eliminierung doppelter Negation

$$\texttt{!(!d)} \quad \Longrightarrow \quad \texttt{d}$$

de Morgan Regeln

$$\texttt{!(}d_1\ \texttt{\&}\ d_2\texttt{)} \quad \Longrightarrow \quad \texttt{(!}d_1\texttt{)}\ \texttt{|}\ \texttt{(!}d_2\texttt{)}$$

$$\texttt{!(}d_1\ \texttt{|}\ d_2\texttt{)} \quad \Longrightarrow \quad \texttt{(!}d_1\texttt{)}\ \texttt{\&}\ \texttt{(!}d_2\texttt{)}$$

Distributionsregeln

$$\texttt{(}d_1\ \texttt{|}\ d_2\texttt{)}\ \texttt{\&}\ d_3 \quad \Longrightarrow \quad \texttt{(}d_1\ \texttt{\&}\ d_3\texttt{)}\ \texttt{|}\ \texttt{(}d_2\ \texttt{\&}\ d_3\texttt{)}$$

9.2.2 Disjunktionsreduktion

Die Reduktion von Disjunktion hilft, redundante Teilformeln zu erkennen. Auf diese Weise wird die spätere Verarbeitung vereinfacht.

Es sei an dieser Stelle darauf hingewiesen, dass im Folgenden Typprüfungen stets mit Hilfe ihrer Interpretation beschrieben werden. Um eine weitere Komplizierung zu vermeiden, wird auf eine syntaktische Beschreibung verzichtet. Die Implementation der Typen durch Bitvektoren (vgl. Abschnitt 11.5.2) ist ohnehin näher an dieser Vorgehensweise.

$$\texttt{c | c} \quad \Longrightarrow \quad \texttt{c}$$

$$\texttt{c | (!c)} \quad \Longrightarrow \quad \top$$

$$\texttt{(!c) | (!c)} \quad \Longrightarrow \quad \texttt{(!c)}$$

$$\texttt{c | t} \quad \Longrightarrow \quad \texttt{t} \quad \text{falls } \texttt{c} \in [\![\texttt{t}]\!]$$

$$\texttt{c | (!t)} \quad \Longrightarrow \quad \begin{cases} \top & \text{falls } [\![\texttt{t}]\!] = \{\texttt{c}\} \\ \texttt{(!t)} & \text{falls } \texttt{c} \notin [\![\texttt{t}]\!] \end{cases}$$

$$\texttt{(!c) | t} \quad \Longrightarrow \quad \begin{cases} \top & \text{falls } [\![\texttt{t}]\!] = \{\texttt{c}\} \\ \texttt{(!c)} & \text{falls } \texttt{c} \notin [\![\texttt{t}]\!] \end{cases}$$

$$\texttt{(!c) | (!t)} \quad \Longrightarrow \quad \begin{cases} \texttt{(!c)} & \text{falls } \texttt{c} \in [\![\texttt{t}]\!] \\ \top & \text{falls } \texttt{c} \notin [\![\texttt{t}]\!] \end{cases}$$

$$\texttt{t}_1\ \texttt{|}\ \texttt{t}_2 \quad \Longrightarrow \quad \begin{cases} \texttt{t}_2 & \text{falls } [\![\texttt{t}_1]\!] \subseteq [\![\texttt{t}_2]\!] \\ \texttt{t}_1 & \text{falls } [\![\texttt{t}_2]\!] \subset [\![\texttt{t}_1]\!] \end{cases}$$

$$\texttt{t}_1 \texttt{ | (!t}_2\texttt{)} \quad \Longrightarrow \quad \begin{cases} \top & \text{falls } [\![\texttt{t}_2]\!] \subseteq [\![\texttt{t}_1]\!] \\ \texttt{(!t}_2\texttt{)} & \text{falls } [\![\texttt{t}_1]\!] \cap [\![\texttt{t}_2]\!] = \emptyset \end{cases}$$

$$\texttt{(!t}_1\texttt{) | (!t}_2\texttt{)} \quad \Longrightarrow \quad \begin{cases} \texttt{(!t}_1\texttt{)} & \text{falls } [\![\texttt{t}_1]\!] \subseteq [\![\texttt{t}_2]\!] \\ \texttt{(!t}_2\texttt{)} & \text{falls } [\![\texttt{t}_2]\!] \subset [\![\texttt{t}_1]\!] \\ \top & \text{falls } [\![\texttt{t}_1]\!] \cap [\![\texttt{t}_2]\!] = \emptyset \end{cases}$$

9.2.3 Konsistenzcheck

Der Konsistenzcheck prüft alle syntaktisch möglichen konjunktiven Verknüpfungen von Literalen auf eine Inkonsistenz ab. Im Falle einer Inkonsistenz ersetzt das $\bot$-Symbol die Teilformel.

Reduktion von Konstanten-Paaren

$$\texttt{c}_1 \texttt{ \& c}_2 \quad \Longrightarrow \quad \begin{cases} \texttt{c}_1 & \text{falls } \texttt{c}_1 = \texttt{c}_2 \\ \bot & \text{sonst} \end{cases}$$

$$\texttt{c}_1 \texttt{ \& (!c}_2\texttt{)} \quad \Longrightarrow \quad \begin{cases} \texttt{c}_1 & \text{falls } \texttt{c}_1 \neq \texttt{c}_2 \\ \bot & \text{sonst} \end{cases}$$

$$\texttt{(!c) \& (!c)} \quad \Longrightarrow \quad \texttt{(!c)}$$

Typ-Check

$$\texttt{c \& t} \quad \Longrightarrow \quad \begin{cases} \texttt{c} & \text{falls } \texttt{c} \in [\![\texttt{t}]\!] \\ \bot & \text{sonst} \end{cases}$$

$$\texttt{c \& (!t)} \quad \Longrightarrow \quad \begin{cases} \texttt{c} & \text{falls } \texttt{c} \notin [\![\texttt{t}]\!] \\ \bot & \text{sonst} \end{cases}$$

$$\texttt{(!c) \& t} \quad \Longrightarrow \quad \texttt{t} \quad \text{falls } \texttt{c} \notin [\![\texttt{t}]\!]$$

$$\texttt{(!c) \& (!t)} \quad \Longrightarrow \quad \texttt{(!t)} \quad \text{falls } \texttt{c} \in [\![\texttt{t}]\!]$$

Reduktion von Typ-Paaren

$$\texttt{t}_1 \texttt{ \& t}_2 \quad \Longrightarrow \quad \begin{cases} \texttt{t}_1 & \text{falls } [\![t_1]\!] \subseteq [\![t_2]\!] \\ \texttt{t}_2 & \text{falls } [\![\texttt{t}_2]\!] \subset [\![\texttt{t}_1]\!] \\ \bot & \text{sonst} \end{cases}$$

$$\texttt{t}_1 \texttt{ \& (!t}_2\texttt{)} \quad \Longrightarrow \quad \begin{cases} \texttt{t}_1 & \text{falls } [\![\texttt{t}_2]\!] \cap [\![\texttt{t}_1]\!] = \emptyset \\ \bot & \text{falls } [\![\texttt{t}_1]\!] \subseteq [\![\texttt{t}_2]\!] \end{cases}$$

$$\texttt{(!t}_1\texttt{) \& (!t}_2\texttt{)} \quad \Longrightarrow \quad \begin{cases} \texttt{(!t}_2\texttt{)} & \text{falls } [\![\texttt{t}_1]\!] \subseteq [\![\texttt{t}_2]\!] \\ \texttt{(!t}_1\texttt{)} & \text{falls } [\![\texttt{t}_2]\!] \subset [\![\texttt{t}_1]\!] \end{cases}$$

9.2.4 Korrektheit und Vollständigkeit

Um den Zusammenhang des Kalküls zur denotationellen Semantik zu illustrieren, werden in diesem Unterabschnitt exemplarisch die Korrektheit und Vollständigkeit der Verarbeitung nachgewiesen. Die Nachweise erfolgen dabei nicht formal, sondern lediglich als Beweisskizze.

Lemma 1 (Korrektheit)
Falls $\mathtt{d} \stackrel{*}{\Longrightarrow} \bot$ *, so gilt* $[\![\mathtt{d}]\!] = \emptyset$.

Beweis: (Skizze)
Weise nach, dass alle Verarbeitungsregeln die Semantik unverändert lassen. Führt dann eine Verarbeitung für ein `d` zu einem Clash, so folgt stets $[\![\mathtt{d}]\!] = \emptyset$.

Der Nachweis erfolgt hier stellvertretend für die Formel $(\mathtt{t}_1\ \mathtt{\&}\ (\mathtt{!}\mathtt{t}_2))$. Da Typen hierarchisch organisiert sind, gibt es drei Fälle. Es wird jeweils nachgewiesen, dass die Denotation der linken Seite der Regel mit der Denotation der rechten Seite der Regel übereinstimmt.

1. $[\![\mathtt{t}_2]\!] \cap [\![\mathtt{t}_1]\!] = \emptyset$:
   $(C \setminus [\![\mathtt{t}_2]\!]) \cap [\![\mathtt{t}_1]\!] = C \cap [\![\mathtt{t}_1]\!] = [\![\mathtt{t}_1]\!]$
2. $[\![\mathtt{t}_1]\!] \subseteq [\![\mathtt{t}_2]\!]$:
   $(C \setminus [\![\mathtt{t}_2]\!]) \cap [\![\mathtt{t}_1]\!] = (C \setminus [\![\mathtt{t}_2]\!]) \cap [\![\mathtt{t}_2]\!] = \emptyset$
3. $[\![\mathtt{t}_2]\!] \subset [\![\mathtt{t}_1]\!]$ In diesem Fall verändert die Regel die Formel nicht.

Lemma 2 (Vollständigkeit)
Falls $[\![\mathtt{d}]\!] = \emptyset$ *, so gilt* $\mathtt{d} \stackrel{*}{\Longrightarrow} \bot$.

Beweis: (Skizze)
Zeige, dass jede Formel mit Hilfe der Verarbeitungsregeln in eine semantisch äquivalente Disjunktive Normalform transformiert werden kann.

Ist eine Formel inkonsistent, so ist die entsprechende Disjunktive Normalform inkonsistent. Damit ist jedes Disjunkt inkonsistent. Für jedes Disjunkt muss es demnach mindestens zwei Literale geben, deren Konjunktion inkonsistent ist.

Zeige, dass die Verarbeitungsregeln für konjunktive Verknüpfungen alle syntaktischen Möglichkeiten abdecken und für inkonsistente Verknüpfungen ein Clash erzeugt wird.

Zeige, dass die Verarbeitungsregeln für disjunktive Verknüpfungen einen Clash weiterpropagieren. Damit ist der Nachweis, dass ein Clash abgeleitet werden kann, erbracht.

9.3 Verarbeitung von Attributspezifikationen

9.3.1 Disjunktive Normalform

Dieser Block stellt eine Disjunktive Normalform auf der Attributspezifikationsebene her. Die entstehenden Disjunktionen werden später auf der Knotenebene nach außen getragen, d.h. durch zwei disjunktive Knotenbeschreibungen repräsentiert. Man beachte, dass Disjunktionen auf Attribut-Wert-Ebene unangetastet bleiben. Dies ist beabsichtigt, um als Ergebnis des Normalisierungsvorgangs nicht zu viele Disjunktionen zu erhalten.

Die Funktion `evalre(z)` wertet einen regulären Ausdruck `z` aus und gibt als Resultat eine Disjunktion von Konstanten $(c_1|\ldots|c_n)$ zurück. Die Domäne einer Attributspezifikation ist `T`, `NT` oder `FREC`.

Reduktion negierter Attribute (t_1 ist die Domäne von `f`)

$$!(f = d) \implies (!t_1) \mid (f = !d)$$

$$!(f \mathrel{!}= d) \implies !(f = !d)$$

$$!(f = /z/) \implies !(f = evalre(z))$$

$$!(f \mathrel{!}= /z/) \implies !(f = !evalre(z))$$

Eliminierung doppelter Negation

$$!(!k) \implies k$$

de Morgan Regeln

$$!(k_1 \,\&\, k_2) \implies (!k_1) \mid (!k_2)$$

$$!(k_1 \mid k_2) \implies (!k_1) \,\&\, (!k_2)$$

Distributionsregeln

$$(k_1 \mid k_2) \,\&\, k_3 \implies (k_1 \,\&\, k_3) \mid (k_2 \,\&\, k_3)$$

9.3.2 Disjunktionsreduktion

Die Reduktion von Disjunktionen hilft, Redundanzen zu erkennen und vereinfacht so die weitere Verarbeitung. Als Typen werden `T`, `NT` und `FREC` verwendet. Das Konzept lässt aber eine spätere Verallgemeinerung offen.

$$t_1 \mid t_2 \implies \begin{cases} t_2 & \text{falls } [\![t_1]\!] \subseteq [\![t_2]\!] \\ t_1 & \text{falls } [\![t_2]\!] \subset [\![t_1]\!] \end{cases}$$

$$t_1 \mid (!t_2) \implies \begin{cases} \top & \text{falls } [\![t_2]\!] \subseteq [\![t_1]\!] \\ (!t_2) & \text{falls } [\![t_1]\!] \cap [\![t_2]\!] = \emptyset \end{cases}$$

$$\texttt{(!t}_1\texttt{) | (!t}_2\texttt{)} \quad \Longrightarrow \quad \begin{cases} \texttt{(!t}_1\texttt{)} & \text{falls } [\![\texttt{t}_1]\!] \subseteq [\![\texttt{t}_2]\!] \\ \texttt{(!t}_2\texttt{)} & \text{falls } [\![\texttt{t}_2]\!] \subset [\![\texttt{t}_1]\!] \\ \top & \text{falls } [\![\texttt{t}_1]\!] \cap [\![\texttt{t}_2]\!] = \emptyset \end{cases}$$

Die folgenden Verarbeitungsblöcke beziehen sich jeweils auf genau ein Disjunkt der Disjunktiven Normalform. Die Disjunkte werden isoliert mit separaten leeren Halden Σ und Θ betrachtet.

9.3.3 Attributnormalisierung

Diese Normalisierungsstufe hat die Aufgabe, Attribut-Wert-Paare in Einträge in der Constraint-Halde zu überführen. Daneben werden reguläre Ausdrücke durch eine disjunktive Aufzählung der erfüllenden Konstanten ersetzt.

$\texttt{f = d} \quad \Longrightarrow \quad \texttt{f = \#x}$

Σ := $\Sigma \cup \{\langle \texttt{\#x}, \texttt{d}\rangle\}$, #x freie Variable

$\texttt{f = \#x} \quad \Longrightarrow \quad \texttt{t}_1\texttt{ \& f=\#x}$

$\Sigma := (\Sigma \setminus \{\langle \texttt{deref(\#x)}, \texttt{d}\rangle\}) \cup$

$\{\langle \texttt{deref(\#x)}, \texttt{(Constant \& t}_2\texttt{ \& d)}\rangle\}$

$\texttt{t}_1$ Domäne, $\texttt{t}_2$ Wertebereich von f

$\texttt{f = \#x:d}_1 \quad \Longrightarrow \quad \texttt{f = \#x}$

$\Sigma := (\Sigma \setminus \{\langle \texttt{deref(\#x)}, \texttt{d}_2\rangle\}) \cup \{\langle \texttt{deref(\#x)}, \texttt{(d}_1\texttt{ \& d}_2\texttt{)}\rangle\}$

$\texttt{f != d} \quad \Longrightarrow \quad \texttt{f = !d}$

$\texttt{f = /z/} \quad \Longrightarrow \quad \texttt{f = evalre(z)}$

$\texttt{f != /z/} \quad \Longrightarrow \quad \texttt{f = !evalre(z)}$

9.3.4 Konsistenzcheck

Dieser Block erkennt und propagiert Inkonsistenzen. Die bei der Reduktion von Attribut-Paaren ggf. entstehenden Inkonsistenzen treten in der Constraint-Halde auf und werden durch die erste Regel auf die oberste Formelebene propagiert.

Constraint-Halde

$\texttt{k} \quad \Longrightarrow \quad \bot \qquad \text{falls } \langle \texttt{\#x}, \bot\rangle \in \Sigma$

Clash-Propagierung

$\bot\texttt{ \& k} \quad \Longrightarrow \quad \bot$

Reduktion von Typ-Paaren

$$\texttt{t}_1\ \texttt{\&}\ \texttt{t}_2 \quad \Longrightarrow \quad \begin{cases} \texttt{t}_1 & \text{falls } [\![t_1]\!] \subseteq [\![t_2]\!] \\ \texttt{t}_2 & \text{falls } [\![\texttt{t}_2]\!] \subset [\![\texttt{t}_1]\!] \\ \bot & \text{sonst} \end{cases}$$

$$\texttt{t}_1\ \texttt{\&}\ \texttt{(!t}_2\texttt{)} \quad \Longrightarrow \quad \begin{cases} \texttt{t}_1 & \text{falls } [\![\texttt{t}_2]\!] \cap [\![\texttt{t}_1]\!] = \emptyset \\ \bot & \text{falls } [\![\texttt{t}_1]\!] \subseteq [\![\texttt{t}_2]\!] \end{cases}$$

$$\texttt{(!t}_1\texttt{)}\ \texttt{\&}\ \texttt{(!t}_2\texttt{)} \quad \Longrightarrow \quad \begin{cases} \texttt{(!t}_2\texttt{)} & \text{falls } [\![\texttt{t}_1]\!] \subseteq [\![\texttt{t}_2]\!] \\ \texttt{(!t}_1\texttt{)} & \text{falls } [\![\texttt{t}_2]\!] \subset [\![\texttt{t}_1]\!] \end{cases}$$

Reduktion von Attribut-Paaren

$$\texttt{(f = \#x}_1\texttt{)}\ \texttt{\&}\ \texttt{(f = \#x}_2\texttt{)} \quad \Longrightarrow \quad \texttt{f = \#x}_1$$
$$\Sigma := (\Sigma \setminus \{\langle \texttt{deref(\#x}_1\texttt{)}, \texttt{d}_1\rangle, \langle \texttt{deref(\#x}_2\texttt{)}, \texttt{d}_2\rangle\}) \cup \{\langle \texttt{deref(\#x}_1\texttt{)}, \texttt{(d}_1\ \texttt{\&}\ \texttt{d}_2\texttt{)}\rangle\}$$
$$\Theta := \Theta \cup \{\langle \texttt{deref(\#x}_2\texttt{)}, \texttt{deref(\#x}_1\texttt{)}\rangle\}$$

9.4 Verarbeitung von Knotenbeschreibungen

Wie im informellen Entwurf der Anfragesprache beschrieben, sind konstante Knoten-IDs nur in Korpus-Graphbeschreibungen zulässig.

9.4.1 Disjunktive Normalform

Die Disjunktionsregel teilt eine Disjunktion innerhalb der Attributspezifikation einer Knotenbeschreibung in zwei Knotenbeschreibungen auf. Damit bleibt die Attributspezifikation disjunktionsfrei. Disjunktionen verbleiben insgesamt nur auf Graphbeschreibungsebene (vgl. nächster Abschnitt) und auf Attribut-Wert-Ebene.

Vorbereiten von Knotenbeschreibungen

$$\texttt{[k]} \quad \Longrightarrow \quad \texttt{[\#x : k]} \qquad \texttt{\#x} \text{ freie Variable}$$

$$\texttt{[\#x}_2\texttt{ : k]} \quad \Longrightarrow \quad \texttt{\#x}_1\texttt{ : [\#x}_2\texttt{ : k]} \qquad \texttt{\#x}_1 \text{ freie Variable}$$

$$\texttt{\#x}_1\texttt{ : [k]} \quad \Longrightarrow \quad \texttt{\#x}_1\texttt{ : [\#x}_2\texttt{ : k]} \qquad \texttt{\#x}_2 \text{ freie Variable}$$

Disjunktion von Attributspezifikationen

$$\texttt{\#x}_1\texttt{ : [\#x}_2\texttt{ : (k}_1\ \texttt{|}\ \texttt{k}_2\texttt{)]} \quad \Longrightarrow \quad \texttt{\#x}_1\texttt{ : [\#x}_2\texttt{ : k}_1\texttt{] | \#x}_1\texttt{ : [\#x}_2\texttt{ : k}_2\texttt{]}$$

Die folgenden Verarbeitungsblöcke beziehen sich jeweils auf genau ein Disjunkt der Disjunktiven Normalform. Die Disjunkte werden isoliert mit separaten leeren Halden Σ und Θ betrachtet.

9.4.2 Normalisierung von Knotenbeschreibungen

Die folgenden Normalisierungsregeln beschreiben den Übergang von Knotenbeschreibungen in die Constraint-Halde.

$$\texttt{c:[k]} \quad \Longrightarrow \quad \texttt{\#x}_1 \qquad \texttt{\#x}_1, \texttt{\#x}_2 \text{ freie Variablen}$$
$$\Sigma := \Sigma \cup \{\langle \texttt{\#x}_1, \langle \texttt{c}, \texttt{\#x}_2\rangle\rangle, \langle \texttt{\#x}_2, \texttt{k}\rangle\}$$

$$\texttt{[ ]} \quad \Longrightarrow \quad \texttt{\#x} \qquad \texttt{\#x} \text{ freie Variable}$$

$$\texttt{\#x:[ ]} \quad \Longrightarrow \quad \texttt{\#x}$$

$$\texttt{\#x}_1\texttt{:[\#x}_2\texttt{]} \quad \Longrightarrow \quad \texttt{\#x}_1 \qquad \langle \texttt{deref(\#x}_1\texttt{)}, \langle \texttt{c}_3, \texttt{\#x}_3\rangle\rangle \in \Sigma$$
$$\Sigma := (\Sigma \setminus \{\langle \texttt{deref(\#x}_2\texttt{)}, \texttt{k}_2\rangle, \langle \texttt{\#x}_3, \texttt{k}_3\rangle\}) \cup \{\langle \texttt{\#x}_3, \texttt{(k}_2 \texttt{ \& k}_3\texttt{)}\rangle\}$$
$$\Theta := \Theta \cup \{\langle \texttt{deref(\#x}_2\texttt{)}, \texttt{\#x}_3\rangle\}$$

$$\texttt{\#x}_1\texttt{:[\#x}_2\texttt{:k}_2\texttt{]} \quad \Longrightarrow \quad \texttt{\#x}_1\texttt{:[\#x}_2\texttt{]}$$
$$\Sigma := (\Sigma \setminus \{\langle \texttt{deref(\#x}_2\texttt{)}, \texttt{k}_3\rangle\}) \cup \{\langle \texttt{deref(\#x}_2\texttt{)}, \texttt{(k}_2 \texttt{ \& k}_3\texttt{)}\rangle\}$$

9.4.3 Knotenauswahl

Die folgende Regel stellt einen zentralen Baustein des Kalküls dar. Sie rät, welcher Korpusgraphknoten $\#\texttt{x}_2$ mit einem Anfrageknoten $\#\texttt{x}_1$ verschmolzen wird. Dazu werden zum einen die Knoten-IDs zusammengefasst, wodurch der Anfrageknoten gebunden wird. Sollte dieser bereits gebunden sein, findet ein Abgleich statt. Ein dadurch ggf. entstehender Clash wird auf der Graphbeschreibungsebene propagiert. Zum anderen werden die Attributspezifikationen in der Constraint-Halde unifiziert. Ein Clash auf dieser Ebene wird durch den Konsistenzcheck auf Knotenbeschreibungsebene propagiert.

In der folgenden Regel wird das $\perp$-Symbol als Knoten-ID verwendet, falls die IDs $\texttt{c}_1$ und $\texttt{c}_2$ nicht übereinstimmen sollten.

$$\texttt{\#x}_1 \quad \Longrightarrow \quad \texttt{\#x}_2$$
für eine Knotenvariable $\texttt{\#x}_1$ in der Anfrage und
für eine Knotenvariable $\texttt{\#x}_2$ im Korpusgraph
$$\Sigma := (\Sigma \setminus \{\langle \texttt{deref(\#x}_1\texttt{)}, \langle \texttt{c}_1, \texttt{\#x}_3\rangle\rangle, \langle \texttt{\#x}_2, \langle \texttt{c}_2, \texttt{\#x}_4\rangle\rangle, \langle \texttt{\#x}_3, \texttt{k}_3\rangle, \langle \texttt{\#x}_4, \texttt{k}_4\rangle\}) \cup \{\langle \texttt{\#x}_2, \langle \texttt{c}_1 \texttt{ \& c}_2, \texttt{\#x}_4\rangle\rangle, \langle \texttt{\#x}_4, \texttt{k}_3 \texttt{ \& k}_4\rangle\}$$
$$\Theta := \Theta \cup \{\langle \texttt{deref(\#x}_1\texttt{)}, \texttt{\#x}_2\rangle, \langle \texttt{deref(\#x}_3\texttt{)}, \texttt{\#x}_4\rangle\}$$

Im Folgenden wird die Gesamtheit aller Knotenauswahl-Schritte für die Variablen einer Anfrage ebenfalls als Knotenauswahl bezeichnet.

9.5 Verarbeitung von Graphbeschreibungen

Nachfolgend wird das $\perp$-Symbol als Graphbeschreibung zugelassen.

9.5.1 Disjunktive Normalform

Diese Regel bringt eine Disjunktion nach ganz außen, so dass insgesamt eine Liste von Disjunktionen entsteht, die isoliert voneinander betrachtet werden können.

Distributionsregel

$$(g_1 \mid g_2) \,\&\, g_3 \quad \Longrightarrow \quad (g_1 \,\&\, g_3) \mid (g_2 \,\&\, g_3)$$

9.5.2 Disjunktionsreduktion

Diese Regel eliminiert den Clash, der durch den Disjunktiven Kontext wirkungslos bleibt.

$$\perp \mid g \quad \Longrightarrow \quad g$$

9.5.3 Normalisierung von Graphbeschreibungen

Da ein Korpusgraph nur aus Basisrelationen besteht, müssen alle abgeleiteten Relationen zunächst aus diesen Relationen hergeleitet werden. Ansonsten wäre ein Test auf Nicht-Basisrelationen nicht durchführbar.

Eliminierung trivialer Knotenrelationen

$$\#x \,\&\, g \quad \Longrightarrow \quad g$$

Normalisierung von Knotenrelationen

$$\#x_1 > \#x_2 \quad \Longrightarrow \quad \#x_1 >1\ \#x_2$$

$$\#x_1 . \#x_2 \quad \Longrightarrow \quad \#x_1 .1\ \#x_2$$

$$\#x_1 >l\ \#x_2 \quad \Longrightarrow \quad (\#x_1 >l\ \#x_2) \,\&\, (\#x_1 >1\ \#x_2)$$

$$\#x_1\ \$.*\ \#x_2 \quad \Longrightarrow \quad (\#x_1\ \$\ \#x_2) \,\&\, (\#x_1\ .*\ \#x_2)$$

Transitive Hülle von Korpusgraphrelationen

$$(\#x_1 >a\ \#x_2) \,\&\, (\#x_2 >b\ \#x_3) \quad \Longrightarrow$$
$$(\#x_1 >a\ \#x_2) \,\&\, (\#x_2 >b\ \#x_3) \,\&\, (\#x_1 >(a+b)\ \#x_3)$$

$$(\#x_1 .a\ \#x_2) \,\&\, (\#x_2 .b\ \#x_3) \quad \Longrightarrow$$
$$(\#x_1 .a\ \#x_2) \,\&\, (\#x_2 .b\ \#x_3) \,\&\, (\#x_1 .(a+b)\ \#x_3)$$

`#x` $\Longrightarrow$ `tokenarity(#x,b)`

`b` ist die Anzahl der Token, die von #x dominiert werden

`#x` $\Longrightarrow$ `arity(#x,b)`

`b` ist die Anzahl der Knoten, die von #x direkt dominiert werden

`#x` $\Longrightarrow$ `continuous(#x)`

wenn #x ein kontinuierlicher Knoten ist

`#x` $\Longrightarrow$ `discontinuous(#x)`

wenn #x ein diskontinuierlicher Knoten ist

`#x` $\Longrightarrow$ `root(#x)`

wenn #x der Wurzelknoten des Korpusgraphen ist

Die folgenden Verarbeitungsblöcke beziehen sich jeweils auf genau ein Disjunkt der Disjunktiven Normalform. Die Disjunkte werden isoliert mit separaten leeren Halden Σ und Θ betrachtet.

9.5.4 Konsistenzcheck

Diese Regeln propagieren einen Clash in der Constraint-Halde und erlauben den Test eines Prädikats bzw. einer Relation gegen einen Korpusgraph.

Constraint-Halde

`g` $\Longrightarrow$ $\bot$ falls $\langle$`#x`$, \bot\rangle$ oder $\langle$`#x`$_1, \langle\bot,$ `#x`$_2\rangle\rangle \in \Sigma$

Clash-Propagierung

$\bot$ `&` `g` $\Longrightarrow$ $\bot$

Stelligkeit (`arity(#x,b')` im Korpusgraph)

`arity(#x,a)` $\Longrightarrow$ $\begin{cases} \texttt{\#x} & \text{falls } \texttt{a} = \texttt{b'} \\ \bot & \text{falls } \texttt{a} \neq \texttt{b'} \end{cases}$

`arity(#x,a,b)` $\Longrightarrow$ $\begin{cases} \texttt{\#x} & \text{falls } \texttt{a} \leq \texttt{b'} \leq \texttt{b} \\ \bot & \text{falls } \texttt{b'} < \texttt{a or b'} > \texttt{b} \end{cases}$

Token-Stelligkeit (`tokenarity(#x,b')` im Korpusgraph)

`tokenarity(#x,a)` $\Longrightarrow$ $\begin{cases} \texttt{\#x} & \text{falls } \texttt{a} = \texttt{b'} \\ \bot & \text{falls } \texttt{a} \neq \texttt{b'} \end{cases}$

`tokenarity(#x,a,b)` $\Longrightarrow$ $\begin{cases} \texttt{\#x} & \text{falls } \texttt{a} \leq \texttt{b'} \leq \texttt{b} \\ \bot & \text{falls } \texttt{b'} < \texttt{a or b'} > \texttt{b} \end{cases}$

Weitere Prädikate

`continuous(#x)` $\Longrightarrow$ `#(x)`

falls continuous(#x) im Korpusgraph

`discontinuous(#x)` $\Longrightarrow$ `#(x)`

falls discontinuous(#x) im Korpusgraph

`root(#x)` $\Longrightarrow$ `#(x)`

falls root(#x) im Korpusgraph

Knotenrelationen

`#x >a #x` $\Longrightarrow$ $\bot$

$\#x_1$ `>l` $\#x_2$ $\Longrightarrow$ $\#x_2$

falls (`#x`$_1$ `>l #x`$_2$) im Korpusgraph

$\#x_1$ `>*` $\#x_2$ $\Longrightarrow$ $\#x_1$

falls (`#x`$_1$ `>a #x`$_2$) im Korpusgraph

$\#x_1$ `>a` $\#x_2$ $\Longrightarrow$ $\#x_1$

falls (`#x`$_1$ `>a #x`$_2$) im Korpusgraph

$\#x_1$ `>a,b` $\#x_2$ $\Longrightarrow$ $\#x_1$

falls (`#x`$_1$ `>a' #x`$_2$) im Korpusgraph, `a` $\leq$ `a'` $\leq$ `b`

$\#x_1$ `>@l` $\#x_2$ $\Longrightarrow$ $\#x_1$

falls `#x`$_2$ linke Ecke von `#x`$_1$ ist

$\#x_1$ `>@r` $\#x_2$ $\Longrightarrow$ $\#x_1$

falls `#x`$_2$ rechte Ecke von `#x`$_1$ ist

`#x . #x` $\Longrightarrow$ $\bot$

$\#x_1$ `.*` $\#x_2$ $\Longrightarrow$ $\#x_1$

falls (`#x`$_1$ `.a #x`$_2$) im Korpusgraph

$\#x_1$ `.a` $\#x_2$ $\Longrightarrow$ $\#x_1$

falls (`#x`$_1$ `.a #x`$_2$) im Korpusgraph

$\#x_1$ `.a,b` $\#x_2$ $\Longrightarrow$ $\#x_1$

falls (`#x`$_1$ `.a' #x`$_2$) im Korpusgraph, `a` $\leq$ `a'` $\leq$ `b`

$\#x_1$ `$` $\#x_2$ $\Longrightarrow$ $\#x_1$

falls (`#x`$_0$ `>` `#x`$_1$), (`#x`$_0$ `>` `#x`$_2$) im Korpusgraph, `#x`$_1$ $\neq$ `#x`$_2$

$\#x_1$ `!>l` $\#x_2$ $\Longrightarrow$ $\#x_2$

falls (`#x`$_1$ `>l #x`$_2$) nicht im Korpusgraph

`#x1 !> #x2` $\Longrightarrow$ `#x2`

falls `∀l (#x1 >l #x2)` nicht im Korpusgraph

`#x1 !>* #x2` $\Longrightarrow$ `#x1`

falls `∀a (#x1 >a #x2)` nicht im Korpusgraph

`#x1 !>a #x2` $\Longrightarrow$ `#x1`

falls `(#x1 >a #x2)` nicht im Korpusgraph

`#x1 !>a,b #x2` $\Longrightarrow$ `#x1`

falls `(#x1 >a' #x2)` im Korpusgraph, `a'` $<$ `a` oder `b` $<$ `a'`

`#x1 !>@l #x2` $\Longrightarrow$ `#x1`

falls `#x2` nicht linke Ecke von `#x1` ist

`#x1 !>@r #x2` $\Longrightarrow$ `#x1`

falls `#x2` nicht rechte Ecke von `#x1` ist

`#x1 !$ #x2` $\Longrightarrow$ `#x1`

falls es kein `#x0` im Korpusgraph gibt mit `(#x0 >1 #x1)` und `(#x0 >1 #x2)`

`#x1 !. #x2` $\Longrightarrow$ `#x1` falls `(#x1 . #x2)` nicht im Korpusgraph

`#x1 !.* #x2` $\Longrightarrow$ `#x1`

falls `∀a (#x1 .a #x2)` nicht im Korpusgraph

`#x1 !.a #x2` $\Longrightarrow$ `#x1`

falls `(#x1 .a #x2)` nicht im Korpusgraph

`#x1 !.a,b #x2` $\Longrightarrow$ `#x1`

falls `(#x1 .a' #x2)` im Korpusgraph, `a'` $<$ `a` oder `b` $<$ `a'`

9.6 Ableitbarkeit einer Anfrage

Da der Kalkül die Ableitbarkeit einer Anfrage aus dem Korpus überprüfen soll, ist der Kalkül entsprechend zu erweitern. Dazu wird ein gegen die Anfrage q zu testender Korpusgraph g eines Korpus k geraten:

$k, g \vdash q$, falls $g \vdash q$ für einen Korpusgraphen g

$g \vdash q$, falls für eine Knotenauswahl nicht gilt $g \,\&\, q \stackrel{*}{\Longrightarrow} \bot$

Eine Anfrage q ist also genau dann aus einem Korpus k ableitbar, wenn es einen Korpusgraphen g gibt, aus dem q abgeleitet werden kann. Eine Anfrage q ist wiederum genau dann aus einem Korpusgraphen g ableitbar, wenn es eine Knotenauswahl (d.h. eine Zuordnung von Korpusgraphknoten zu Knotenbeschreibungen der Anfrage) gibt, mit der für $g \,\&\, q$ kein Widerspruch $\perp$ abgeleitet werden kann.

Es sei angemerkt, dass diese Definition implizit dafür sorgt, dass bei einer Knotenauswahl immer **alle** Knotenbeschreibungen der Anfrage mit entsprechenden Graphknoten verschmolzen worden sind. Denn ansonsten wäre nicht sichergestellt, dass nicht doch (durch eine weitere Verschmelzung einer Knotenbeschreibung) ein Widerspruch abgeleitet werden kann.

9.7 Beispielverarbeitung einer Korpusanfrage

Um einen Eindruck zu vermitteln, wie die Verarbeitung des Kalküls konzipiert ist, wird nach der formalen Definition der Verarbeitungsregeln die Verarbeitungsweise an einer Beispielanfrage illustriert.

Datenbasis

Als Datenbasis dient der Korpusgraph in Abb. 9.2. Um das Beispiel übersichtlich zu halten, wird bei der Korpusdefinition lediglich der Teilgraph betrachtet, der die Nominalphrase enthält.

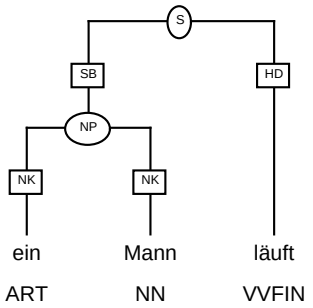


Abbildung 9.2: Der Beispielgraph

Die Definition des Korpusgraphen sieht dann folgendermaßen aus:

```
"t1":[word="ein" & pos="ART"] &
"t2":[word="Mann" & pos="NN"] &
"np":[cat="NP"] &
```

```
"t1" . "t2" &

"np" >NK "t1" &
"np" >NK "t2" &

arity("np",2)
```

Anfrage

Als Beispielanfrage dient die folgende Graphbeschreibung, die offensichtlich vom Beispielgraphen gematcht wird:

```
#phrase:[cat="S" | cat="NP"] & #token:[pos="ART" & word="ein"] &
#phrase > #token
```

Resultierende Graphbeschreibung

Der Kalkül stellt fest, ob es eine Knotenauswahl (d.h. eine Zuordnung von Korpusgraphknoten zu Knotenbeschreibungen der Anfrage) gibt, mit der für `g & q` kein Widerspruch abgeleitet werden kann. In diesem Falle kann die Anfrage `q` aus dem Korpus `k` unter der ermittelten Variablenbelegung abgeleitet werden.

Da im Beispiel das Korpus aus nur einem Korpusgraphen besteht, entfällt das Raten des Korpusgraphen. Die zu untersuchende Graphbeschreibung lautet:

```
(#phrase:[cat="S" | cat="NP"] & #token:[pos="ART" & word="ein"] &
 #phrase > #token) &

("t1":[word="ein" & pos="ART"] &
 "t2":[word="Mann" & pos="NN"] &
 "np":[cat="NP"] &
 "t1" . "t2" &
 "np" >NK "t1" & "np" >NK "t2" &
 arity("np",2))
```

Die Beispielverarbeitung wird nun Schritt für Schritt beschrieben.

1. Schritt: Disjunktive Normalform

Zunächst lässt sich feststellen, dass der Korpusgraph bereits in Disjunktiver Normalform vorliegt (vgl. Unterabschnitt 6.2.7). Das Herstellen der Normalform konzentriert sich daher zunächst auf die Korpusanfrage, bevor Definition und Anfrage in einer gemeinsamen Disjunktiven Normalform zusammengefasst werden.

Die Anwendung der Regeln zur Disjunktiven Normalform auf Knotenbeschreibungsebene (vgl. Unterabschnitt 9.4.1) führt dazu, dass zunächst die Attribut-Spezifikations-Variablen `#f1` und `#f2` für beide Knotenbeschreibungen eingefügt werden. Anschließend wird die Disjunktion in der Attributspezifikation des Knotens `#phrase` nach außen gezogen:

$\Longrightarrow$

```
#phrase:[#f1: cat="S" | cat="NP"] &
#token:[#f2: pos="ART" & word="ein"] &
#phrase > #token &
"t1":[word="ein" & pos="ART"] & "t2":[word="Mann" & pos="NN"] &
"np":[cat="NP"] &
"t1" . "t2" & "np" >NK "t1" & "np" >NK "t2" & arity("np",2)
```

$\Longrightarrow$

```
(#phrase:[#f1: cat="S"] | #phrase:[#f1: cat="NP"]) &
(#token:[#f2: pos="ART" & word="ein"] &
 #phrase > #token &
 "t1":[word="ein" & pos="ART"] & "t2":[word="Mann" & pos="NN"] &
 "np":[cat="NP"] &
 "t1" . "t2" & "np" >NK "t1" & "np" >NK "t2" & arity("np",2))
```

$\Longrightarrow$

```
(#phrase:[#f1: cat="S"] &
 #token:[#f2: pos="ART" & word="ein"] &
 #phrase > #token &
 "t1":[word="ein" & pos="ART"] & "t2":[word="Mann" & pos="NN"] &
 "np":[cat="NP"] &
 "t1" . "t2" & "np" >NK "t1" & "np" >NK "t2" & arity("np",2)
)
|
(#phrase:[#f1: cat="NP"]) &
 #token:[#f2: pos="ART" & word="ein"] &
 #phrase > #token &
 "t1":[word="ein" & pos="ART"] & "t2":[word="Mann" & pos="NN"] &
 "np":[cat="NP"] &
 "t1" . "t2" & "np" >NK "t1" & "np" >NK "t2" & arity("np",2)
)
```

Auf diese Weise entstehen zwei disjunkte Graphbeschreibungen, die nun völlig isoliert voneinander verarbeitet werden. Stellvertretend wird die Verarbeitung des zweiten Disjunkts fortgesetzt. Die Verarbeitung des ersten Disjunkts führt durch das Fehlen eines `S`-Knotens im Korpusgraphen nicht zum Erfolg.

2. Schritt: Belegen der Constraint-Halde

Als weiterer Schritt im Rahmen der Anfragenormalisierung werden zunächst freie Variablen an die Attributwerte gebunden (vgl. Attributnormalisierung in Unterabschnitt 9.3.3). Beispielsweise wird der Ausdruck `cat="NP"` im Knoten `#phrase` zu `cat=#d1:"NP"` erweitert.

Anschliessend werden die beiden Anfrageknoten `#phrase` und `#np` und alle Graphknoten `"t1"`, `"t2"` und `"np"` normalisiert (vgl. DNF von Knotenbeschreibungen in Unterabschnitt 9.4.1) und ihre Spezifikationen in die Constraint-Halde eingewiesen (vgl. Normalisierung von Knotenbeschreibungen in Unterabschnitt 9.4.2). Die entstandene Constraint-Halde ist nach Knotenbeschreibungen, Attributspezifikationen und Attributwerten gruppiert. Die einzelnen Einträge sind dabei zur besseren Lesbarkeit in einer vereinfachten Abbildungsschreibweise notiert.

```
#phrase:[#f1: cat="NP"] &
#token:[#f2: pos="ART" & word="ein"] &
#phrase > #token &
"t1":[word="ein" & pos="ART"] & "t2":[word="Mann" & pos="NN"] &
"np":[cat="NP"] &
"t1" . "t2" & "np" >NK "t1" & "np" >NK "t2" & arity("np",2)
```

$\Longrightarrow$

```
#phrase:[#f1: cat=#d1:"NP"]) &
#token:[#f2: pos=#d2:"ART" & word=#d3:"ein"] &
#phrase > #token &
"t1":[#f3: word=#d5:"ein" & pos=#d4:"ART"] &
"np":[#f4: cat=#d6:"NP"] &
"t2":[#f5: word=#d7:"Mann" & pos=#d8:"NN"] &
"t1" . "t2" & "np" >NK "t1" & "np" >NK "t2" & arity("np",2)
```

$\Longrightarrow$

```
#phrase &
#token &
#phrase > #token &
#t1 &
#np &
#t2 &
"t1" . "t2" & "np" >NK "t1" & "np" >NK "t2" & arity("np",2)
```

Constraint-Halde:

`#phrase` $\rightarrow$ <$\top$,`#f1`>, `#token` $\rightarrow$ <$\top$,`#f2`>,
`#t1` $\rightarrow$ <`"t1"`,`#f3`>, `#np` $\rightarrow$ <`"np"`,`#f4`>,

#t2 → <⊤,#f5>
#f1 → cat=#d1, #f2 → pos=#d2 & word=#d3,
#f3 → pos=#d4 & word=#d5, #f4 → cat=#d6,
#f5 → word=#d7 & pos=#d8
#d1 → "NP", #d2 → "ART", #d3 → "ein",
#d4 → "ART", #d5 → "ein", #d6 → "NP",
#d7 → "Mann", #d8 → "NN"

3. Schritt: Raten von Knotenverschmelzungen

Es gilt nun zu prüfen, ob die beiden Anfrageknoten mit zwei in einer direkten Dominanzbeziehung stehenden Graphknoten verknüpft werden können. Für den menschlichen Betrachter ist dabei unmittelbar einsehbar, welche Auswahl von Graphknoten zum Erfolg führt. Für die Implementation ist dies hingegen ein entscheidender Punkt, der die Effizienz der Anfrageverarbeitung maßgeblich beeinflusst.

An dieser Stelle kommt die zentrale Regel des Kalküls zur Anwendung (vgl. Unterabschnitt 9.4.3). Diese Regel bestimmt nichtdeterministisch die nächste Knotenverschmelzung. Hier fasst sie zunächst die Knoten `#phrase` und `#np` zusammen. Durch die Zusammenfassung werden die zugehörigen Attributspezifikationen `#f1` und `#f4` in der Constraint-Halde durch eine konjunktive Verknüpfung unifiziert. Ob die beiden Spezifikationen überhaupt unifizierbar sind, klärt der spätere Konsistenzcheck. Dabei wird zum ersten Mal die Variablenumbenennungs-Halde verwendet.

⟹

```
#token &
#phrase > #token &
#t1 &
#np &
#t2 &
"t1" . "t2" & "np" >NK "t1" & "np" >NK "t2" & arity("np",2)
```

Constraint-Halde:

#token → <⊤,#f2>,
#t1 → <"t1",#f3>, #np → <⊤ & "np",#f4>,
#t2 → <⊤,#f5>
#f2 → pos=#d2 & word=#d3,
#f3 → pos=#d4 & word=#d5, #f4 → cat=#d1 & cat=#d6,
#f5 → word=#d7 & pos=#d8
#d1 → "NP", #d2 → "ART", #d3 → "ein",
#d4 → "ART", #d5 → "ein", #d6 → "NP",
#d7 → "Mann", #d8 → "NN"

Variablenumbenennungs-Halde:
#phrase → #np, #f1 → #f4

⟹

```
#token &
#phrase > #token &
#t1 &
#np &
#t2 &
"t1" . "t2" & "np" >NK "t1" & "np" >NK "t2" & arity("np",2)
```

Constraint-Halde:
#token → <⊤,#f2>,
#t1 → <"t1",#f3>, #np → <"np",#f4>,
#t2 → <⊤,#f5>
#f2 → pos=#d2 & word=#d3,
#f3 → pos=#d4 & word=#d5, #f4 → cat=#d6,
#f5 → word=#d7 & pos=#d8
#d2 → "ART", #d3 → "ein",
#d4 → "ART", #d5 → "ein", #d6 → "NP" & "NP",
#d7 → "Mann", #d8 → "NN"
Variablenumbenennungs-Halde:
#phrase → #np, #f1 → #f4, #d6 → #d1

Für den Attributwert #d6 entsteht in der Constraint-Halde die Verknüpfung "NP" & "NP". Die anschließende Anwendung des Konsistenzchecks von Attributwerten (vgl. Unterabschnitt 9.2.3) auf den Attributwert #d6 führt zur Konstanten "NP". Die Verschmelzung der beiden Knoten ist damit erfolgreich.

Die Verschmelzung der beiden Knoten #token und #t1 verläuft analog und führt dann zu folgendem Zwischenergebnis:

$\stackrel{*}{\Longrightarrow}$

```
#phrase > #token &
#t1 &
#np &
#t2 &
"t1" . "t2" & "np" >NK "t1" & "np" >NK "t2" & arity("np",2)
```

Constraint-Halde:
#t1 → <"t1",#f3>, #np → <"np",#f4>,
#t2 → <⊤,#f5>
#f3 → pos=#d4 & word=#d5, #f4 → cat=#d6,
#f5 → word=#d7 & pos=#d8
#d4 → "ART", #d5 → "ein", #d6 → "NP",

#d7 → "Mann", #d8 → "NN"

Variablenumbenennungs-Halde:

#phrase → #np, #token → #t1,
#f1 → #f4, #f2 → #f3,
#d2 → #d4, #d3 → #d5, #d6 → #d1

4. Schritt: Konsistenzcheck der Graphbeschreibung

In der resultierenden Graphbeschreibung sind die beiden Anfrageknoten nun gebunden. Die einzig verbleibende Möglichkeit einer Inkonsistenz liegt in der Relation `#phrase > #token`, die aus der Anfrage stammt. Dazu wird abschließend ein Konsistenzcheck auf Graphbeschreibungsebene durchgeführt (vgl. Unterabschnitt 9.5.4):

Der Korpusgraph verwendet die Basisrelation beschriftete Dominanz. Um einen Test der direkten Dominanzbeziehung durchführen zu können, muss zunächst eine Relationsableitung innerhalb der Graphdefinition erfolgen (vgl. Normalisierung von Graphbeschreibungen in Unterabschnitt 9.5.3). Im Beispiel wird aus `"np" >NK "t1"` die Relation `"np" >1 "t1"` abgeleitet. Man kann diese Ableitung auch als Erweiterung der Datenbasis um ein weiteres Faktum auffassen.

$\Longrightarrow$

```
#phrase >1 #token &
#t1 &
#np &
#t2 &
"t1" . "t2" &
"np" >NK "t1" & "np" >1 "t1" &
"np" >NK "t2" & arity("np",2)
```

Constraint-Halde:

#t1 → <"t1",#f3>, #np → <"np",#f4>,
#t2 → <⊤,#f5>
#f3 → pos=#d4 & word=#d5, #f4 → cat=#d6,
#f5 → word=#d7 & pos=#d8
#d4 → "ART", #d5 → "ein", #d6 → "NP",
#d7 → "Mann", #d8 → "NN"

Variablenumbenennungs-Halde:

#phrase → #np, #token → #t1,
#f1 → #f4, #f2 → #f3,
#d2 → #d4, #d3 → #d5, #d6 → #d1

Da die Variable `#phrase` an den Knoten `"np"` und die Variable `#token` an den Knoten `"t1"` gebunden ist (vgl. Constraint-Halde und Umbenennungs-Halde),

und zudem die Basisrelation `"np" >1 "t1"` als Faktum vorliegt, fällt der Konsistenzcheck positiv aus (vgl. Unterabschnitt 9.5.4).

Als Ergebnis des Checks der Knotenrelation `#phrase > #token` verbleibt nach Definition die Variable `#token` (vgl. Unterabschnitt 9.5.4). Sie wird durch die Eliminierung der trivialen Knotenrelation entfernt (vgl. Normalisierung von Graphbeschreibungen in Unterabschnitt 9.5.3). Als Ergebnis ergibt sich also:

$\Longrightarrow$

```
#t1 &
#np &
#t2 &
"t1" . "t2" &
"np" >NK "t1" &  "np" >1 "t1" &
"np" >NK "t2" & arity("np",2)
```

Constraint-Halde:

`#t1` $\rightarrow$ <`"t1",#f3`>, `#np` $\rightarrow$ <`"np",#f4`>,
`#t2` $\rightarrow$ <$\top$,`#f5`>
`#f3` $\rightarrow$ `pos=#d4 & word=#d5`, `#f4` $\rightarrow$ `cat=#d6`,
`#f5` $\rightarrow$ `word=#d7 & pos=#d8`
`#d4` $\rightarrow$ `"ART"`, `#d5` $\rightarrow$ `"ein"`, `#d6` $\rightarrow$ `"NP"`,
`#d7` $\rightarrow$ `"Mann"`, `#d8` $\rightarrow$ `"NN"`

Variablenumbenennungs-Halde:

`#phrase` $\rightarrow$ `#np`, `#token` $\rightarrow$ `#t1`,
`#f1` $\rightarrow$ `#f4`, `#f2` $\rightarrow$ `#f3`,
`#d2` $\rightarrow$ `#d4`, `#d3` $\rightarrow$ `#d5`, `#d6` $\rightarrow$ `#d1`

Ergebnis

An der normalisierten Formel ist abzulesen, dass keine Möglichkeit einer Inkonsistenz (d.h. der Ableitung eines Widerspruchs $\perp$) verbleibt. Die Verarbeitung ist damit erfolgreich. Als Ergebnis kann zunächst festgehalten werden, dass die Anfrage aus dem Korpus abgeleitet werden kann. An den Variablenbelegungen der Constraint-Halde lässt sich darüberhinaus ablesen, an welchen Korpusgraphen und an welche Korpusknoten die Variablen im Einzelnen gebunden wurden:

`#phrase` $\rightarrow$ `"np"`
`#token` $\rightarrow$ `"t1"`

Diese Information ist für die Weiterverarbeitung der Anfrageergebnisse nützlich und wird deshalb beim Export der Ergebnisse im TIGER-XML-Format (vgl. Kapitel 7) und bei der Ergebnisvisualisierung (vgl. Abschnitt 14.4) verwendet.

Kapitel 10

Konzeption der Implementation

Nach der Definition des Verarbeitungskalküls geht es in diesem Kapitel um die Konzeption der Kalkül-Implementation. Dazu werden in Abschnitt 10.1 zunächst die Ansätze der in Kapitel 3 vorgestellten Baumbanksuchwerkzeuge systematisiert und diskutiert. Auf der Grundlage dieser Diskussion legt Abschnitt 10.2 die Architektur der hier verfolgten Implementation fest. Die Bausteine dieser Implementation werden in den nachfolgenden Kapiteln detailliert beschrieben.

10.1 Systematisierung bisheriger Arbeiten

Eine Klassifizierung der Ansätze der verfügbaren Baumbankwerkzeuge ist in Abb. 10.1 zu finden. Diese Klassifizierung wird im Folgenden erläutert und diskutiert. Zwei grundsätzliche Vorgehensweisen sind zu dabei unterscheiden.

Verarbeitung zur Laufzeit

Die erste Vorgehensweise sieht vor, die gesamte Verarbeitung einer Anfrage zur Laufzeit durchzuführen. Es werden keine Daten über das Korpus erhoben, das originale Korpus wird zur Laufzeit eingeladen und verarbeitet (vgl. *CorpusSearch*, Abschnitt 3.3).

Als Vorteil kann festgehalten werden, dass die Anfrageverarbeitung bei dieser Konzeption stets auf der aktuellen Korpusversion erfolgen kann. Dies ist für dynamische Korpora eine wertvolle Eigenschaft. Der offensichtliche Nachteil besteht darin, dass stets das gesamte Korpus eingeladen und verarbeitet werden muss. Selbst wenn aus der Anfrage bereits ersichtlich ist, dass nur ein kleiner Teil des Korpus für einen Match in Frage kommt, wird stets der gesamte Verarbeitungsmechanismus verwendet. Als Konsequenz ist eine solche Vorgehensweise vergleichsweise ineffizient.

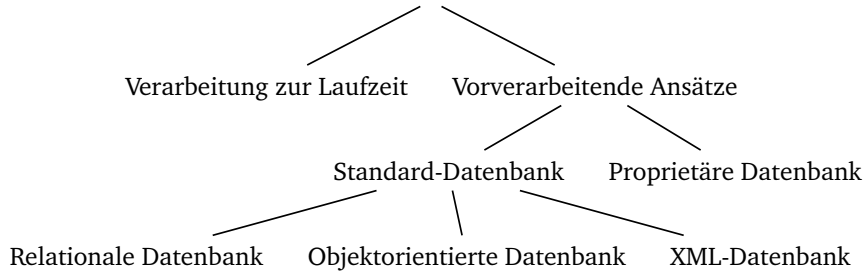


Abbildung 10.1: Systematisierung der Ansätze

Vorverarbeitende Ansätze

Die zweite Vorgehensweise teilt die Verarbeitung in zwei Schritte. In einem Vorverarbeitungsschritt wird ein Korpus eingelesen und überarbeitet. Dabei wird es zum einen in eine andere Form gebracht – beispielsweise in ein internes Datenbank-Format oder XML. Zum anderen werden Daten über das Korpus gesammelt, die zur effizienten Auswertung späterer Anfragen hilfreich sind. Hier werden also bereits Suchvorgänge durchgeführt, deren Ergebnisse die Verarbeitung bestimmter Anfragetypen erleichtern werden. Als Beispiel sei die Berechnung der transitiven Hülle der direkten Dominanzrelation eines Korpusgraphen genannt, der die spätere Auswertung einer allgemeinen Dominanzbeziehung effizienter macht. Eine explizite Anreicherung des originalen Korpus um implizit enthaltene Daten wird als Indexierung bezeichnet.

In einem separaten Schritt erfolgt die Verarbeitung von Korpusanfragen. Sie greift auf die gewonnenen Daten zurück. Durch die Formatkonvertierung sind die Korpusdaten meist schneller zugreifbar, durch die Indexierung sind Teilanfragen (wie z.B. die allgemeine Dominanzbeziehung) schneller auswertbar.

Der Vorteil der schnelleren Anfrageverarbeitung wird durch den Nachteil erkauft, dass für jedes neue Korpus stets eine Vorverarbeitung erfolgen muss. Wie stark dieser zusätzliche Aufwand ins Gewicht fällt, hängt von der Dynamik des Korpus ab. Ist ein Korpus statisch, ist der Aufwand zu vernachlässigen. Für ein dynamisches Korpus ist eine Vorverarbeitung hingegen unpraktikabel.

Die meisten verfügbaren Suchwerkzeuge sehen die Vorverarbeitung eines Korpus vor. Die Unterschiede zwischen den Ansätzen liegen in der technischen Verwaltung der in der Vorverarbeitung erzeugten Korpusdaten.

Vorverarbeitung unter Verwendung einer Standard-Datenbank

Eine naheliegende Vorgehensweise zur Verwaltung der erzeugten Korpusdaten besteht in der Verwendung einer Datenbank. Bei der Vorverarbeitung wird das Korpus eingelesen, ggf. um weitere Informationen angereichert und dann in eine Datenbank überführt (vgl. *VIQTORYA*, Abschnitt 3.5).

Datenbanken stellen eine ausgereifte Technologie dar, die effizient und zuverlässig arbeitet. Die Auswahl des genauen Datenbanktyps (relationale Datenbank, objektorientierte Datenbank, XML-Datenbank) mag zwar technische Vor- und Nachteile nach sich ziehen (Verfügbarkeit von Anfragesprachen, Verfügbarkeit standardisierter Programmierschnittstellen), der konzeptionelle Vorteil einer Datenbanklösung liegt aber auf weniger auf der technischen Ebene.

Der große Vorteil einer solchen Lösung besteht vielmehr darin, dass die Implementation einer Verarbeitung von Korpusanfragen entfällt. Datenbanken bieten unabhängig vom Typ bereits eine Verarbeitung für Anfragen an, die in einer Datenbankanfragesprache (z.B. SQL für relationale Datenbanken, XML Query für XML-Datenbanken) formuliert werden können. Da diese Anfragesprachen für die computerlinguistischen Anwender eines Anfragewerkzeugs wenig geeignet sind, sehen Systeme wie *VIQTORYA* eine linguistisch motivierte Anfragesprache als Front-End vor. Suchanfragen, die in dieser Anfragesprache formuliert sind, werden dann in die Anfragesprache der Datenbank transformiert.

Der Implementationsaufwand einer Datenbank-Lösung ist durch die Verfügbarkeit einer Anfrageverarbeitung gering. Auf der anderen Seite ist eine Datenbank eine notwendige Voraussetzung für ein solches Anfragewerkzeug. Die Verfügbarkeit einer Datenbank ist aber nicht immer gewährleistet und eine Distribution des Sucherkzeugs muss demnach zumindest optional eine vollwertige Datenbank enthalten.

Vorverarbeitung unter Verwendung einer proprietären Datenbank

Die Alternative besteht darin, eine proprietäre Datenbank zu verwenden. Die Korpusdaten werden also in ein selbst entwickeltes (meist binäres) Format transformiert (vgl. *tgrep*, Abschnitt 3.2). Diese Daten werden bei der Auswertung von Korpusanfragen wieder eingelesen und zur effizienten Verarbeitung eingesetzt. Damit ist diese Lösung nicht auf eine vorhandene Datenbank angewiesen, die proprietäre Datenbanklösung ist in der Software enthalten. Ein weiterer Vorteil besteht in der Offenheit des Konzepts, die das System flexibel und skalierbar macht.

Der Nachteil besteht im deutlich höheren Entwicklungsaufwand, da die Verarbeitung von Anfragen an die proprietäre Datenbank selbst entwickelt werden muss. Dieser Nachteil kann allerdings auch als Chance verstanden werden, da die proprietäre Lösung keinem vorgegebenen Schema folgt (z.B. Tabellenstruktur in relationalen Datenbanken), sondern genau auf das vorliegende Datenmodell zugeschnitten werden kann.

10.2 Architektur der Implementation

Zwei Anforderungen stehen bei der Entwicklung des Baumbankanfrage-Werkzeugs in dieser Arbeit im Vordergrund: Zum einen die effiziente Verarbeitung großer Korpora, zum anderen die benutzerfreundliche, auf den Adressatenkreis zugeschnittene Gestaltung des Software-Pakets.

Soll eine effiziente Verarbeitung gewährleistet werden, führt an einer Vorverarbeitung des Korpus kein Weg vorbei. Bei der Entscheidung für oder gegen eine Standard-Datenbank-Lösung sollte bedacht werden, dass auch die Gestaltung der Distribution, also die Installation der Software, über die Benutzerfreundlichkeit einer Software entscheidet. Da eine proprietäre Datenbank auf jedem System lauffähig ist, ist für die vorliegende Arbeit die proprietäre Lösung gewählt worden.

Die weitere Strukturierung dieses Teils der Arbeit ergibt sich unmittelbar aus der resultierenden Systemarchitektur (vgl. Abb. 10.2). Zentraler Bestandteil dieser Architektur ist die proprietäre Datenbank, die in Kapitel 11 beschrieben wird. Sie wird im Folgenden kurz als Index bezeichnet, da die Gewinnung möglichst umfassender Zusatzinformation über das Korpus im Mittelpunkt steht. Bezogen auf den TIGER-Kalkül nimmt der Index möglichst viele Verarbeitungsschritte wie die Ableitung von Relationsbeziehungen aus den Basisrelationen für die Korpusgraphen vorweg und stellt die Ergebnisse dieser Ableitungen schnell zugreifbar zur Verfügung.

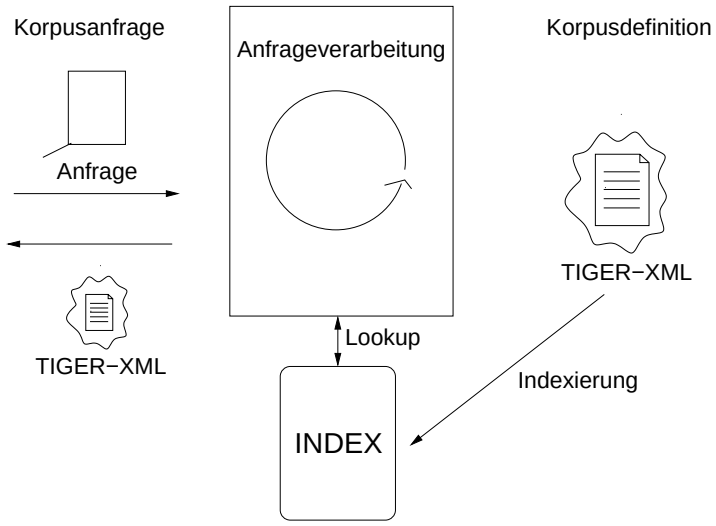


Abbildung 10.2: Architektur der Implementation

Die eigentliche Anfrageverarbeitung umfasst damit zur Laufzeit idealerweise nur noch diejenigen Verarbeitungsschritte des Kalküls, die die Korpusanfrage verarbeiten und damit nicht vorab durchgeführt werden können. Kapi-

tel 12 zeigt auf, wie die Verarbeitung dieser verbleibenden Kalkülregeln realisiert wird. Ein Schwerpunkt liegt dabei auf der Umsetzung der beiden nichtdeterministischen Verarbeitungsstufen.

Das abschließende Kapitel 13 befasst sich mit der Effizienzsteigerung der Implementation. Hier werden Indexerweiterungen vorgenommen und Techniken zur Anfrageoptimierung und zur Filterung des Suchraums eingesetzt. Zudem werden technische Ansätze verfolgt, die die Eigenschaften der verwendeten Programmiersprache Java ausnutzen.

Kapitel 11

Korpusrepräsentation im Index

Das vorliegende Kapitel beschreibt die Repräsentation des Korpus als Index. Es zeigt auf, wie die Korpusdefinitions-Daten in eine schnell zugreifbare Form gebracht werden und damit eine proprietäre Korpus-Datenbank bilden. Zwar werden auch in diesem Rahmen bereits Daten gesammelt, die über die exakte Definition der Korpusgraphen hinausgehen. Die Notwendigkeit für erweiterte Korpusdaten, die speziell auf die Beschleunigung bestimmter Teilsuchen zugeschnitten sind, wird sich jedoch erst aus der Implementation der Anfrageverarbeitung zur Laufzeit ergeben (vgl. nachfolgendes Kapitel).

Der Korpusindex stellt eine effizient zugreifbare binäre Kodierung der textuellen Korpusrepräsentation dar. Er teilt sich in fünf Ebenen, die in den folgenden Abschnitten 11.1 bis 11.5 beschrieben werden: Knotenbeschreibungen, Prädikate, Knotenrelationen, Graphen und das Korpus. Die im Index gesammelten Daten repräsentieren die Ergebnisse der Korpusdefinition und der mittels Kalkülregeln abgeleiteten Aussagen über das Korpus (vgl. Kalkül-Verarbeitung von Graphbeschreibungen in den Unterabschnitten 9.5.3 und 9.5.4).

Da es in diesem Kapitel um die realisierten Ideen und nicht um die Details der Implementation geht, bleibt die Beschreibung stets auf einer abstrakten Ebene. Bei den meisten realisierten Ideen handelt es sich um Standard-Methoden des Information Retrieval. Eine Übersicht über diese Methoden ist in (Baeza-Yates und Ribeiro-Neto 1999, Kapitel 8) zu finden. Zudem orientiert sich die Verwaltung der Korpusdaten in Teilen an der internen Verarbeitung des Anfrageprozessors *CQP* (vgl. Rooth und Christ 1994).

11.1 Knotenbeschreibungen

Die elementaren Bausteine der Korpusgraphen und damit auch des gesamten Korpus sind die Graphknoten. Die Modellierung eines Knoten in Form einer Knotenbeschreibung ist als geordnetes Paar bestehend aus der Knoten-ID und der Spezifikation der Attribute realisiert. Die ersten beiden Unterabschnitte

11.1.1 und 11.1.2 beschreiben die Verwaltung der Attributspezifikation. Hier werden die Attributwerte zunächst numerisch kodiert und dann als Liste von Nummern abgespeichert. Der nachfolgende Unterabschnitt 11.1.3 zeigt, wie die Kantenbeschriftungen als zusätzliches Attribut realisiert werden. Diese Vorgehensweise ist nicht unbedingt naheliegend und wird daher an dieser Stelle diskutiert. Abschließend stellt Unterabschnitt 11.1.4 die Verwaltung der Knoten-IDs dar.

Die einzelnen Abschnitte sind wie folgt gegliedert: Zunächst werden die zu repräsentierenden Daten analysiert und die sich daraus ergebende Datenstruktur vorgestellt. Anschließend folgt eine Auflistung der Zugriffsoperationen, die diese Datenstruktur für die Anfrageauswertung zur Verfügung stellt. Die Operationen sind in einer Java-ähnlichen Notation festgehalten. Als Abschluss wird jeweils der Algorithmus skizziert, der im Rahmen des Indexierungsprozesses die beschriebene Datenstruktur aus den ursprünglichen Korpusdaten gewinnt.

11.1.1 Attributwerte

Datenstruktur

Eine Hauptaufgabe des Index ist die Repräsentation des Korpus und damit insbesondere aller Korpusknoten. Ein Korpus besteht prinzipiell aus einer Aneinanderreihung von Knoten, über denen in einer separaten Schicht Dominanz- und Präzedenzbeziehungen definiert werden. Korpusknoten bestehen hier aus der ID und den Attributwerten für die in der Korpusdeklaration definierten Attribute.

Es ist dabei unmittelbar einsichtig, dass die Attributwerte nicht als Zeichenketten abgelegt werden. Da viele Attribute nur eine begrenzte Anzahl von Werten annehmen (z.B. Wortarten-Tags), führt eine Zeichenketten-Angabe pro Attribut zu einem enorm hohen Speicherbedarf. Stattdessen werden die Attributwerte nummeriert.

Wie Abb. 11.1 am Beispiel eines Attributs `pos` zeigt (`pos` steht hier für die Wortart), werden dabei zwei Richtungen des Zugriffs unterstützt. Zum einen werden alle verwendeten Attributwerte in einer Liste verwaltet[1]. Damit lässt sich zu einer gegebenen Nummer der zugehörige Attributwert bestimmen. Für bestimmte Teile der Anfrageverarbeitung (z.B. Suchraumfilter, vgl. Kapitel 13.2) wird zusätzlich die umgekehrte Operation benötigt. Dazu werden alle Attributwert-Nummer-Paare als Schlüssel-Wert-Paare einer Hashtabelle verwaltet. Die Hashtabelle garantiert eine effiziente Ermittlung der Nummerierung eines Attributwerts und erlaubt zusätzlich den Test, ob eine gegebene Zeichenkette einen verwendeten Attributwert darstellt.

[1] Die Auflistung muss nicht notwendigerweise in alphabetischer Reihenfolge geschehen.

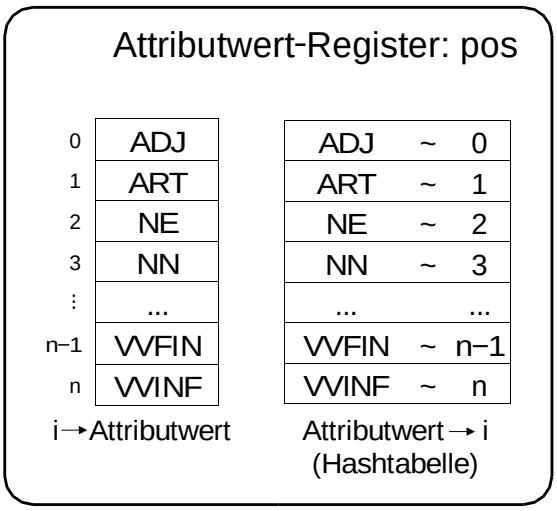


Abbildung 11.1: Ein Register für Attributwerte

Die beiden Datenstrukturen bilden zusammen ein sogenanntes Attribut-Register. Für jedes Attribut des Korpus wird ein solches Register geführt.

Zugriffsoperationen

Auf der Datenstruktur sind folgende Zugriffsoperationen definiert:

- `boolean isFeatureValue(String feature, String value)`

  Prüft, ob die Zeichenkette `value` Attributwert eines Attributs mit dem Namen `feature` ist.

- `int getFeatureValueNumber(String feature, String value)`

  Ermittelt zu einem gegebenen Attributwert `value` eines Attributs mit dem Namen `feature` die zugehörige Nummer.

- `String getFeatureValue(String feature, int number)`

  Ermittelt zu einer gegebenen Nummer `number` eines Attributs mit dem Namen `feature` den zugehörigen Attributwert.

- `String[] getFeatureValues(String feature)`

  Ermittelt zu einem Attribut mit dem Namen `feature` alle zugehörigen Attributwerte.

Konstruktion

Bei der Konstruktion der Attribut-Register kommt die Korpusdeklaration zum Tragen, da sie alle Attribute eines Korpus vollständig auflistet. Die Attributwerte könnten ebenfalls aus der Korpusdeklaration abgelesen werden, allerdings ist ihre Auflistung in der Deklaration optional. Stattdessen wird bei der Indexierung immer dann eine neue Nummer in einem Attributregister vergeben, wenn ein neuer Attributwert im Korpus verwendet wird (vgl. nachfolgenden Abschnitt). Die Attributdeklaration dient dabei als Quelle für Konsistenz-Prüfungen. Ein Attributwert ist neu, wenn er noch nicht als Schlüssel in der Hashtabelle eingetragen ist. Ist er aber nicht deklariert, so handelt es sich um einen Annotationsfehler. Dieser Fehler wird bei der Indexierung als Fehlermeldung nach außen getragen.

11.1.2 Attributspezifikation

Datenstruktur

Da auf dieser Ebene nur die Knoten selbst und keine Knotenrelationen betrachtet werden, kann das Korpus zunächst als eine Liste von Knoten verstanden werden. Da in einem Korpus für terminale und nicht-terminale Knoten sinnvollerweise unterschiedliche Attribute deklariert werden (vgl. Korpusdeklaration in Abschnitt 5.2), werden auch im Index Attributspezifikationen für terminale und nicht-terminale Knoten unterschieden.

Wie Abb. 11.2 illustriert, werden für jedes Nichtterminal-Attribut und für jedes Terminal-Attribut die Attributbelegungen aller Korpusknoten durch ihre Nummern als Liste repräsentiert. Die Listenlängen m bzw. n geben also die Anzahl aller Nichtterminal-Knoten bzw. Terminalknoten im Korpus an.

Die Position eines Listeneintrags entspricht dabei der Nummer des Knotens im Korpus. Während bei den Terminalknoten die Reihenfolge der Knotennummern die Reihenfolge der Token im Satz widerspiegelt (entscheidend für die Präzedenzrelation, vgl. Unterabschnitt 11.3.2), ist die Reihenfolge der Nichtterminalknoten unerheblich. Sinnvollerweise werden die Terminal- und Nichtterminalknoten eines Graphen zu einem Block zusammengefasst. Die Kodierung des Graphanfangs und -endes ist in Abschnitt 11.4 beschrieben.

In der Abbildung werden als Beispiel die nicht-terminalen Attribute `cat` (sytaktische Kategorie) sowie die terminalen Attribute `pos` (Wortart) und `word` (Wortform) bei der Verwaltung der Attributwerte der Korpusknoten verwendet. Durch die Kombination der Attributnummern mit dem Attributregister lässt sich nun feststellen, welcher Attributwert für welchen Korpusknoten definiert ist. Im Beispiel lauten die Wortarten der ersten drei Token: `ADJ ART NN`.

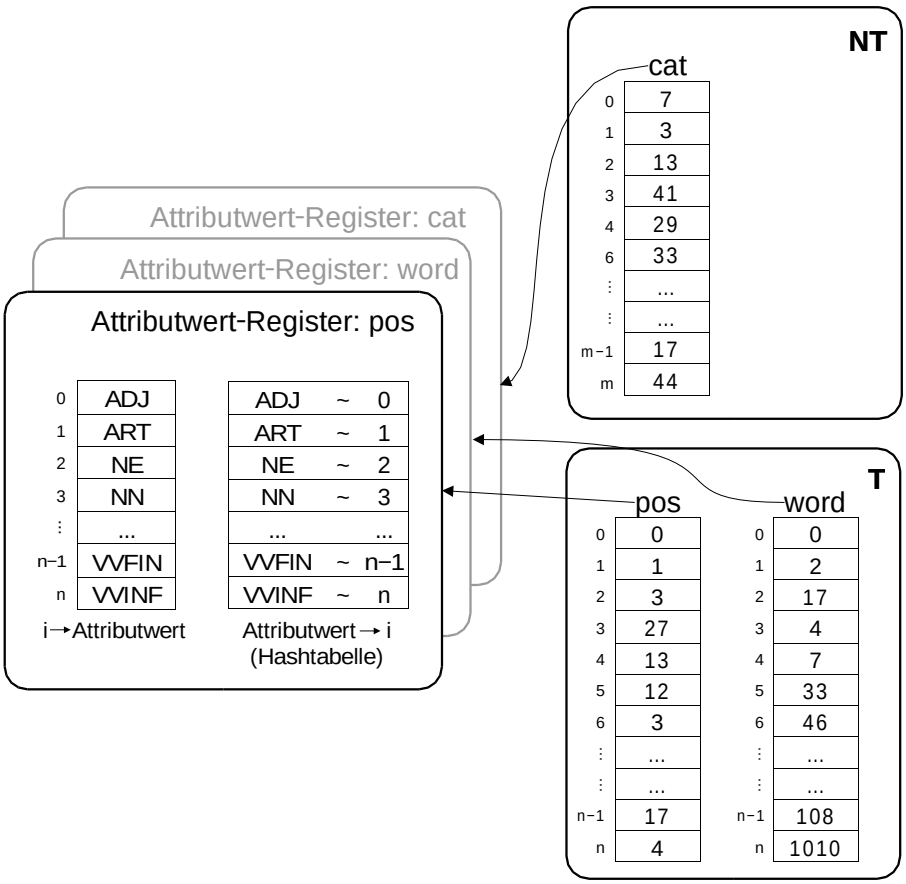


Abbildung 11.2: Die Attributspezifikation der Korpusknoten

Zugriffsoperationen

- `int getTFeatureValueAt(String feature, int position)`

  Ermittelt die Nummer des Attributwerts für das Attribut mit dem Namen `feature` für den terminalen Knoten an der Position `position`.

- `int getNTFeatureValueAt(String feature, int position)`

  Ermittelt die Nummer des Attributwerts für das Attribut mit dem Namen `feature` für den nicht-terminalen Knoten an der Position `position`.

Konstruktion

Bei der Indexierung eines Korpus wird die Textrepräsentation des Korpus, d.h. die TIGER-XML-Kodierung (vgl. Kapitel 7), graphenweise von vorn nach hinten durchlaufen. Damit werden die Attributwerte der Korpusknoten ebenfalls Graph für Graph abgelegt.

Da die Terminale eines Graphen angeordnet sind, kann ihre Reihenfolge unmittelbar übernommen werden. Die Nichtterminale werden ebenfalls in der Reihenfolge ihrer Definition bearbeitet und nummeriert. Für jedes für Terminal- bzw. Nichtterminalknoten deklarierte Attribut wird nun die Nummer des Attributwerts im Attributregister abgelesen. Sollte der Attributwert neu sein (vgl. vorangegangenen Abschnitt), wird im Attributregister eine neue Nummer vergeben und als Ergebnis zurückgegeben. Diese Nummer wird nun in das entsprechende Attributfeld übertragen. Im Falle einer Inkonsistenz wird der gesamte Korpusgraph ignoriert und eine entsprechende Fehlermeldung ausgegeben.

Wesentlich für den benötigten Speicherbedarf ist der elementare Datentyp, der zur Kodierung einer Nummer verwendet wird. In der derzeitigen Implementation wird die Anzahl der Attributwerte zu Beginn der Indexierung in der Korpusdeklaration abgelesen und anhand dieser Information der integrale Basistyp *byte*, *short* oder *int* festgelegt (1, 2 oder 4 bytes). Sind die Attributwerte für ein Attribut nicht deklariert (z.B. für das Wortform-Attribut), so wird zunächst der Datentyp *int* gewählt, dieser Datentyp wird aber nach erfolgter Indexierung überprüft und ggf. korrigiert.

Die beschriebene Realisierung ist natürlich nicht optimal, da sie die Ungleichverteilung der Attributwerte (wenige Attributwerte machen meist einen Großteil aller Vorkommen aus) nicht ausnutzt. Eine verbesserte Variante wird deshalb fortgeschrittene Kompressionstechniken verwenden, z.B. die Huffman-Kodierung. Eine Einführung in die wichtigsten Textkomprimierungstechniken ist in (Baeza-Yates und Ribeiro-Neto 1999, Kapitel 7) zu finden.

11.1.3 Kantenbeschriftungen

Da die beschriftete Dominanz eine der beiden Basisrelationen darstellt (vgl. Kapitel 5), ist es naheliegend, dass die Verwaltung der Kantenbeschriftungen bei der Verwaltung der Dominanzbeziehungen erfolgt. Um jedoch den Implementationsaufwand einzuschränken, wird die Kantenbeschriftung `L` einer Kante vom Knoten `v` zum Knoten `w` dem untergeordneten Knoten `w` als Attributwert eines zusätzlich erzeugten Attributs *edge* zugeordnet. D.h. für ein Korpus, das von Kantenbeschriftungen Gebrauch macht (abzulesen an der Korpusdeklaration), wird ein zusätzliches Attribut *edge* definiert, das auf terminalen und nichtterminalen Knoten operiert (vgl. auch Abb. 11.3). Dieses Attribut trägt für jeden Knoten die Beschriftung der eingehenden Kante. Für den Wurzelknoten eines Graphen ist die eingehende Kantenbeschriftung als undefiniert markiert.

Datenstruktur

Die resultierende Datenstruktur stellt eine Erweiterung der bisherigen Datenstruktur dar. Abb. 11.3 zeigt das zusätzliche Attributregister sowie die Attributwertlisten für das neue Attribut *edge* für terminale und nicht-terminale Knoten.

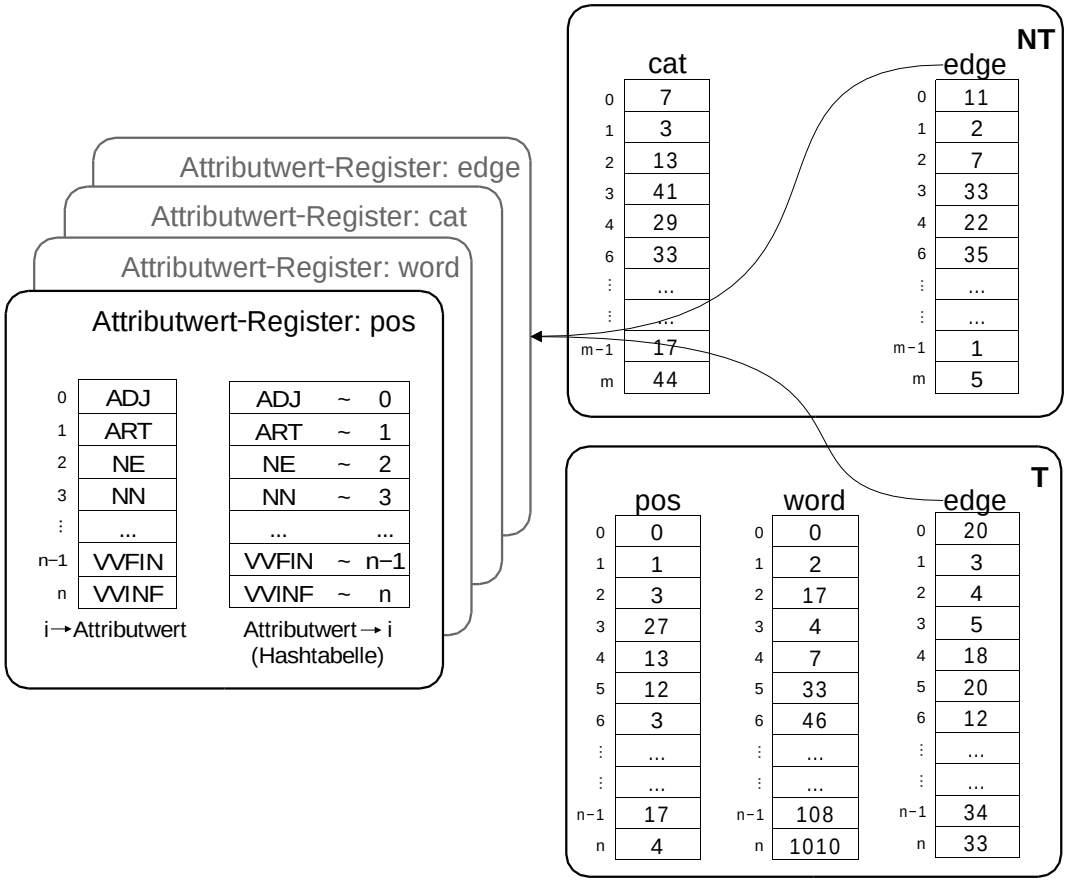


Abbildung 11.3: Kantenbeschriftungen als Attribut *edge*

Der Attributname *edge* stellt durch diese Lösung für die Repräsentation der Kantenbeschriftungen einen reservierten Attributnamen dar. Diese Einschränkung ist aber als unwesentlich einzustufen.

Zugriffsoperationen

- `int getTFeatureValueAt("edge", int position)`

  Ermittelt die Nummer der Beschriftung der eingehende Kante für den terminalen Knoten an der Position `position`.

- `int getNTFeatureValueAt("edge", int position)`

  Ermittelt die Nummer der Beschriftung der eingehende Kanten für den nicht-terminalen Knoten an der Position `position`.

Konstruktion

Die Erzeugung des Attributregisters sowie der Attributlisten verläuft wie in den beiden vorangegangenen Abschnitten für Korpusattribute beschrieben.

11.1.4 Knoten-IDs

Als verbleibender Bestandteil einer Knotenbeschreibung wird nun die Verwaltung der Knoten-IDs erläutert. Dazu muss zunächst geklärt werden, welche Rolle die in der Korpusdefinition festgelegten Knoten-IDs überhaupt spielen sollen.

Zunächst einmal scheint es sinnvoll, dass beim Export von Anfrageergebnissen in Form eines TIGER-XML-Dokuments die ursprünglichen Knoten-IDs des Eingangskorpus verwendet werden. Da Knoten-IDs innerhalb von Korpusbeschreibungen nur zur Korpusdefinition und nicht zur Korpusanfrage verwendet werden, können statt der ursprünglichen IDs problemlos auch Ersatz-IDs verwendet werden.

Zur Vereinfachung der Verarbeitung werden deshalb bei der internen Repräsentation statt der Knoten-IDs die Knoten-Nummern verwendet, die bereits bei der Attribut-Spezifikation eingeführt worden sind (vgl. Unterabschnitt 12.3.1). Die terminalen und nicht-terminalen Knoten eines Korpus werden demnach durch die Nummerierung $0, \ldots, m$ bzw. $0, \ldots, n$ plus Knotendomäne (terminal bzw. nicht-terminal) unterschieden.

Datenstruktur

Bedingt durch diese Vereinfachung werden die ursprünglichen Knoten-IDs nur noch beim Export von Ergebnissen benötigt. Sie werden deshalb parallel zu den Korpusknoten zeilenweise in einer Textdatei abgelegt (vgl. Abb. 11.4). Die Einschränkung, dass die IDs in der Textdatei nur sequentiell eingelesen werden, ist insofern sekundär, als der Export von Anfrageergebnissen auch sequentiell nach der Anordnung der Graphen im Korpus erfolgt.

Zugriffsoperationen

- `int getNTNodeID(int position)`

  Ermittelt die ID des nicht-terminalen Knotens an der Position `position`.

- `int getTNodeID(int position)`

  Ermittelt die ID des terminalen Knotens an der Position `position`.

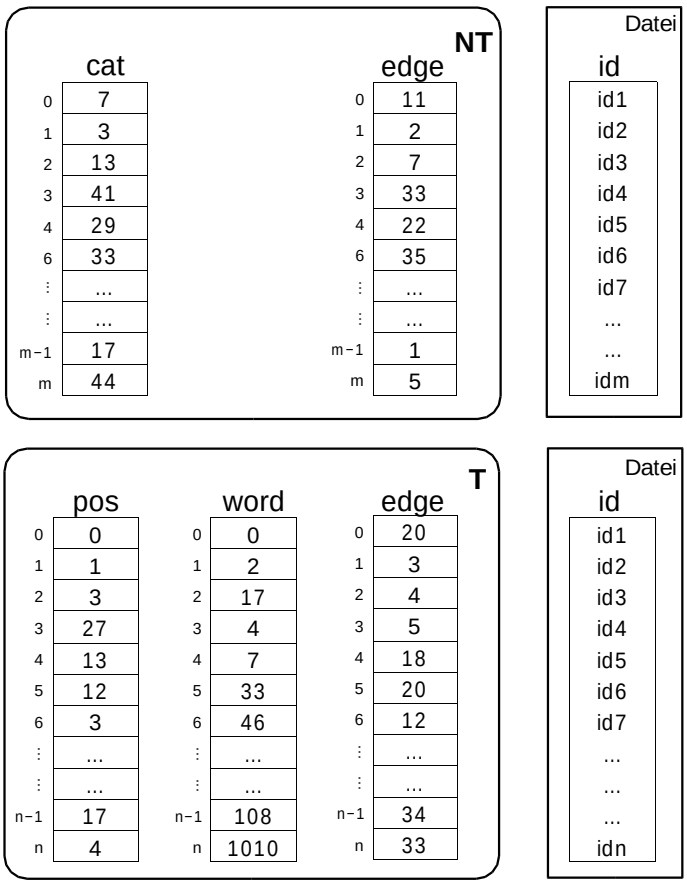


Abbildung 11.4: Verwaltung der Knoten-IDs

Konstruktion

Die Knoten-IDs werden in der selben Reihenfolge wie die Attributwerte der Korpusknoten in die Textdateien geschrieben.

11.2 Prädikate

Nachdem die Verwaltung der Knotenbeschreibungen behandelt wurde, geht es im Folgenden um zentrale Eigenschaften der Knoten, die mittels der Knotenprädikate `continuous`, `discontinuous`, `arity` und `tokenarity` abgefragt werden können. Das Prädikat `root` wird dagegen mit der Graphstruktur kodiert (vgl. Abschnitt 11.4).

Für die Knotenprädikate kann argumentiert werden, dass kein Handlungbedarf besteht, da diese Informationen schließlich zur Laufzeit bestimmt werden können. Doch gerade diese Eigenschaften stellen einen Ansatzpunkt dar, an dem der indexbasierte Ansatz seine Stärken ausspielt. Gerade weil das Nach-

prüfen der Kontinuität und auch der Tokenstelligkeit recht kostspielig ist, ist es sinnvoll, diese Tests schon während der Indexierung durchzuführen und die Ergebnisse im Index direkt zugreifbar zu machen.

Für die Stelligkeit ist dieses Argument nicht unmittelbar einzusehen, da zu vermuten ist, dass die Töchter eines Knotens ebenfalls im Index gespeichert werden und die Anzahl der Töchter damit direkt abzulesen ist. Der nachfolgende Abschnitt wird jedoch zeigen, dass die explizite Kodierung der Töchter auch Nachteile hat und deshalb nicht verfolgt wird (vgl. Abschnitt 11.3). Stattdessen wird die Stelligkeit für jeden inneren Knoten ebenfalls an dieser Stelle kodiert.

Datenstruktur

Stelligkeit, Token-Stelligkeit und Kontinuität werden nun parallel zu den Attributwerten in drei separaten Listen verwaltet (vgl. Abb. 11.5). Für terminale Knoten `#t` sind die folgenden Prädikatwerte vordefiniert und demnach redundant: `arity(#t,1)`, `tokenarity(#t,1)`, `continuous(#t)`.

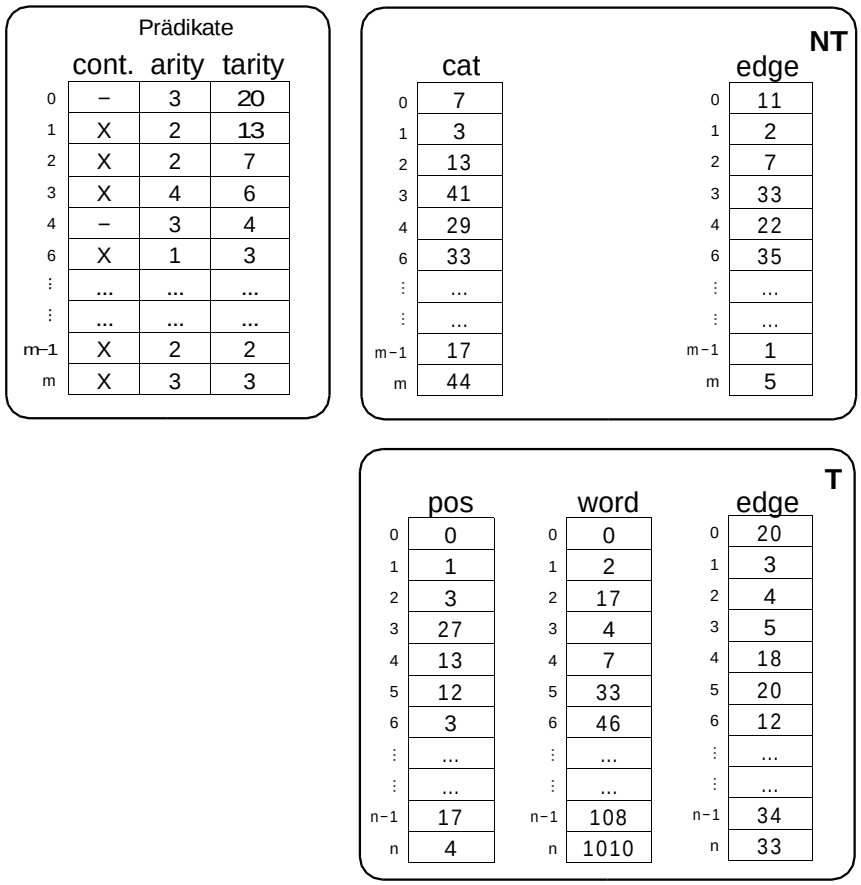


Abbildung 11.5: Kodierung von Knoteneigenschaften für Prädikat-Tests

Die Werte für die Stelligkeit und Token-Stelligkeit weisen eine typische Ungleichverteilung auf. Es ist beispielsweise sehr wahrscheinlich, dass Werte für die Stelligkeiten zwischen 1 und 3 dominant sind. Hier besteht wieder ein großes Potenzial für Kompressionsverfahren.

Zugriffsoperationen

- `int getArity(int position)`

  Ermittelt die Stelligkeit des nicht-terminalen Knotens an der Position `position`.

- `int getTokenArity(int position)`

  Ermittelt die Token-Stelligkeit des nicht-terminalen Knotens an der Position `position`.

- `boolean isContinuous(int position)`

  Prüft, ob der nicht-terminale Knotens an der Position `position` kontinuierlich ist.

Konstruktion

- Kontinuität eines inneren Knotens

  Berechne die Projektion des Knotens auf die Token-Ebene rekursiv als die Vereinigungsmenge der Projektionen seiner Töchter. Die Projektion eines Terminalknotens mit der Nummer t ist dabei als die Menge $\{t\}$ definiert.

  Ordne die Nummern in der Projektionsmenge aufsteigend als Liste. Gibt es in dieser Nummernliste nur Abstände der Länge 1, so ist die Projektion des Knotens auf die Terminalebene nicht durchbrochen. Der Knoten ist nach Definition kontinuierlich, andernfalls diskontinuierlich (vgl. Definition in Unterabschnitt 5.1.3).

- Token-Stelligkeit eines inneren Knotens

  Berechne die Token-Stelligkeit eines Knotens rekursiv als die Summe der Token-Stelligkeiten seiner Töchter. Die Token-Stelligkeit eines Terminalknotens wird als 1 definiert.

- Stelligkeit eines inneren Knotens

  Die Stelligkeit eines inneren Knotens ist direkt in der Korpusdefinition als die Anzahl der ausgehenden Kanten ablesbar (vgl. TIGER-XML-Darstellung, Kapitel 7).

11.3 Relationen

Nachdem die Verwaltung der Korpusknoten und ihrer Eigenschaften realisiert ist, müssen nun die Relationen zwischen den Knoten auf eine möglichst geschickte Art und Weise repräsentiert werden.

Ein Ansatz, der beispielsweise im *VIQTORYA*-System verfolgt wird (vgl. Abschnitt 3.5), listet alle Knotenpaare eines Korpusgraphen auf und bestimmt für jedes Paar die Gültigkeiten der erforderlichen Relationsbeziehungen: *Besteht eine Dominanzbeziehung? Wenn ja, in wieviel Stufen? Besteht eine Geschwisterbeziehung? usw.* Der Vorteil eines solchen Ansatzes besteht darin, dass jede Relationsbeziehung zwischen zwei Knoten unmittelbar abgelesen werden kann. Der Nachteil liegt im hohen Speicherbedarf. Zwar kann die Fülle an Relationsinformationen durch Klassenbildung auf eine Konstante eingeschränkt werden (vgl. *VIQTORYA-Ansatz*, Abschnitt 3.5), doch wächst die Anzahl der zu betrachtenden Knotenpaare quadratisch mit der Anzahl der Knoten.

Dazu eine Überschlagskalkulation: Ein Satz der TIGER-Baumbank (vgl. Abschnitt 1.2) umfasst im Mittel etwa 17 Terminale (Token) und 8 Nichtterminale. Für diese 25 Knoten müssen also 625 Knotenpaare plus Relationsinformation verwaltet werden. Werden pro Knotenpaar realistische 10 Byte Speicherbedarf veranschlagt, so liegt der Kodierungsaufwand pro Graph nur für die Relationskodierung bei etwa 6 KB.

Auf der Suche nach einem Kompromiss zwischen effizientem Zugriff und kompakter Speicherung wird hier ein anderer Weg beschritten. Für die beiden Basisrelationen Dominanz und Präzedenz wird versucht, genau die Menge an Information im Index abzulegen, die ausreicht, um in vertretbarer Zeit einen Relationstest für zwei Knoten durchzuführen. Die dazu verfolgten Strategien werden in den beiden folgenden Unterabschnitten 11.3.1 und 11.3.2 beschrieben.

11.3.1 Dominanz

Für die Repräsentation der Dominanzbeziehungen wird die so genannte *Gorn-Adressierung* benutzt (vgl. Gorn 1967). Sie wird in der Computerlinguistik u.a. im Rahmen der *Tree Adjoining Grammar* verwendet. Die Idee dieser Adressierung besteht darin, jeden Knoten eines Baumes als eine Zeichenkette a über dem Alphabet $\mathbb{N}$ der natürlichen Zahlen zu repräsentieren, d.h. $a \in \mathbb{N}^*$. Diese Zeichenkette wird als Gorn-Adresse bezeichnet.

Die Vergabe der Adressen ist rekursiv definiert: Der Wurzelknoten bekommt als Adresse die leere Zeichenkette ϵ. Ein Mutterknoten gibt seine Adresse a an seine Töchterknoten wie folgt weiter: Der erste Tochterknoten bekommt die Adresse $a0$, die zweite die Adresse $a1$ usw.

Obwohl ursprünglich für Baumstrukturen gedacht, lässt sich diese Adressierung auch auf das vorliegende Datenmodell übertragen. Die Anordnung der Töchterknoten spielt dabei für die Ausnutzung der Gorn-Adressen zur Dominanzkodierung keine Rolle. Da es sich im vorliegenden Datenmodell um gerichtete, azyklische Graphen mit genau einem Mutterknoten handelt, kann die obige Definition ohne Anpassungen verwendet werden. Durch die Festlegung der Adresse des Wurzelknotens kann die Definition der Adressierung rekursiv auf den gesamten Graphen angewendet werden. Abb. 11.6 zeigt die Gorn-Adressierung an einem Beispielgraphen.

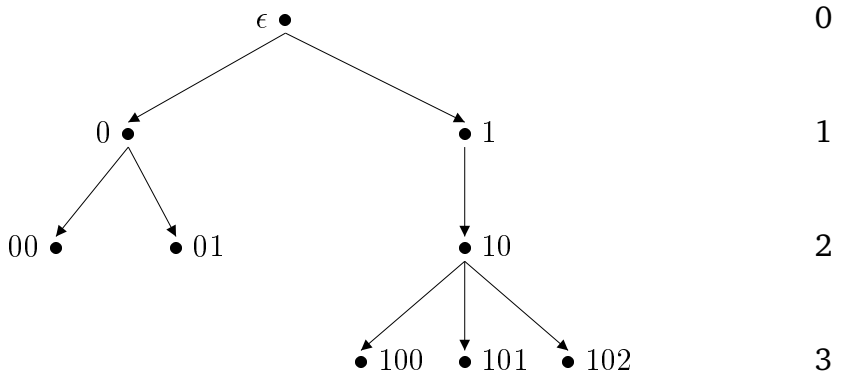


Abbildung 11.6: Gorn-Adressierung

Man beachte, dass die Adressen einer Graphebene (bei top-down-Betrachtung) stets dieselbe Länge haben. Für die Repräsentation der Dominanzbeziehung ist die folgende Aussage von zentraler Bedeutung.

Satz 1 (Dominanzkodierung durch Gorn-Adressierung)
Seien v und w zwei Knoten eines Syntaxgraphen G und seien V bzw. W die Gorn-Adressen der beiden Knoten. Dann gilt:

$$v >n\ w \quad \Leftrightarrow \quad \textit{V ist Präfix von W} \quad \wedge \quad |W| \Leftrightarrow |V| = n$$

Beweis:

a) "$\Rightarrow$"
Es gelte $v >n\ w$, d.h. es gibt einen Pfad p der Länge n in G mit $p = \langle v_0, v_1, \ldots, v_n \rangle$ und $v_0 = v$ und $v_n = w$.
Dann setzt sich nach der Definition der Gorn-Adressierung die Gorn-Adresse W von w aus der Gorn-Adresse V von v und n weiteren Buchstaben des Alphabets zusammen.
D.h. V ist Präfix von W und $|W| \Leftrightarrow |V| = n$.

b) "$\Leftarrow$"
Sei nun V Präfix von W mit $|W| \Leftrightarrow |V| = n$, d.h. $W = V\ w_1 \ldots w_n$.
Seien nun v_i $(0 < i < n)$ die durch die Gorn-Adressen $V_i = V\ w_1 \ldots w_i$ eindeutig bezeichneten Graphknoten. V_i ist eine gültige Gorn-Adresse eines Knotens in G, da ansonsten W keine gültige Gorn-Adresse von w wäre.
Da die Gorn-Adressierung anhand der Mutter-Tochter-Beziehung definiert wird, ist $\langle v_i, v_{i+1} \rangle$ stets eine Kante in G $(0 \leq i < n)$.
Damit ist der Pfad $p = \langle v_0, v_1, \ldots, v_n \rangle$ mit $v_0 = v$ und $v_n = w$ ein Pfad in G.
D.h. $v >n\ w$.

Der folgende Satz stellt eine unmittelbare Folgerung aus Satz 1 dar. Er hält fest, wie die übrigen Relationsbeziehungen zwischen zwei Knoten anhand ihrer Gorn-Adressen abgelesen werden kann. Da die Folgerungen unmittelbar einsichtig sind, werden sie nicht explizit nachgewiesen.

Satz 2 (Kodierung der Dominanzrelationen)
Seien v und w zwei Knoten eines Syntaxgraphen G und seien V bzw. W die Gorn-Adressen der beiden Knoten. Dann gilt:

$$
\begin{array}{rcl}
v\ >m,n\ w & \Leftrightarrow & \textit{V ist Präfix von W} \ \wedge\ |W| \Leftrightarrow |V| = i\ \textit{ mit } m \leq i \leq n. \\
v\ >\ w & \Leftrightarrow & \textit{V ist Präfix von W} \ \wedge\ |W| \Leftrightarrow |V| = 1. \\
v\ >L\ w & \Leftrightarrow & \textit{V ist Präfix von W} \ \wedge\ |W| \Leftrightarrow |V| = 1 \ \wedge \\
 & & \textit{L ist die Kantenbeschriftung der eingehenden Kante von w.} \\
v\ >*\ w & \Leftrightarrow & \textit{V ist Präfix von W.} \\
v\ \$\ w & \Leftrightarrow & \textit{Die Gorn-Adressen von v und w stimmen bis auf} \\
 & & \textit{den letzten Buchstaben überein.}
\end{array}
$$

Es bleibt festzuhalten, dass die Gorn-Adressierung zweier Knoten das Ablesen aller Dominanzbeziehungen erlaubt, die zwischen diesen Knoten gelten können. Die dazu nötigen Operationen Präfixtest und Längendifferenz-Berechnung sind vergleichsweise effizient ausführbar, da sie die Gleichheit integraler Werte überprüfen.

Was den Speicherbedarf der Gorn-Adressierung betrifft, so hängt dieser unmittelbar von der durchschnittlichen Höhe der Korpusgraphen ab. Dieser Parameter variiert abhängig vom Korpus. Wird wieder pro Satz eine Durchschnittslänge von 17 Token und 8 inneren Knoten veranschlagt, so stellt eine mittlere Höhe von 5 Ebenen einen realistischen Wert dar. Eine erste Überschlagsrechnung führt demnach zu etwa 100 Bytes Speicherbedarf pro Graph für die Gornadressierung der Knoten (17 Token mit je 4 Bytes; 8 Knoten mit bis zu 3 Bytes). Rechnet man noch 4 Bytes pro Knoten für die Anbindung der Gorn-Adressen an die Knoten hinzu, entsteht insgesamt ein Speicherbedarf von 200 Bytes. Dieser Wert hat sich beim Praxiseinsatz der TIGERSearch-Software bestätigt. Der Speicherbedarf dieser Lösung ist also als moderat einzustufen.

Datenstruktur

Für die Repräsentation der Gorn-Adressen werden zwei Gorn-Adresslisten angelegt (vgl. Abb. 11.7). Parallel zu den Attributwerten und Knoteneigenschaften wird nun für jeden nicht-terminalen und terminalen Knoten ein Verweis auf seine Gorn-Adresse innerhalb der Adressliste angelegt. Die Adresse ist dann ab der Verweisstelle byte-weise abzulesen. Die Länge der Kodierung kann unmittelbar aus der Differenz zwischen dem nachfolgenden Verweis und dem aktuellen Verweis berechnet werden. Um dem Wurzelknoten eines Graphen die leere Gorn-Adresse zuzuordnen, wird dem nachfolgenden Knoten derselbe Verweis zugewiesen, wodurch sich eine Länge von 0 Buchstaben für die Gorn-Adresse ergibt (vgl. auch Dominanzkodierung des Nichtterminals mit der Nummer 0 im Beispiel in Abb. 11.7).

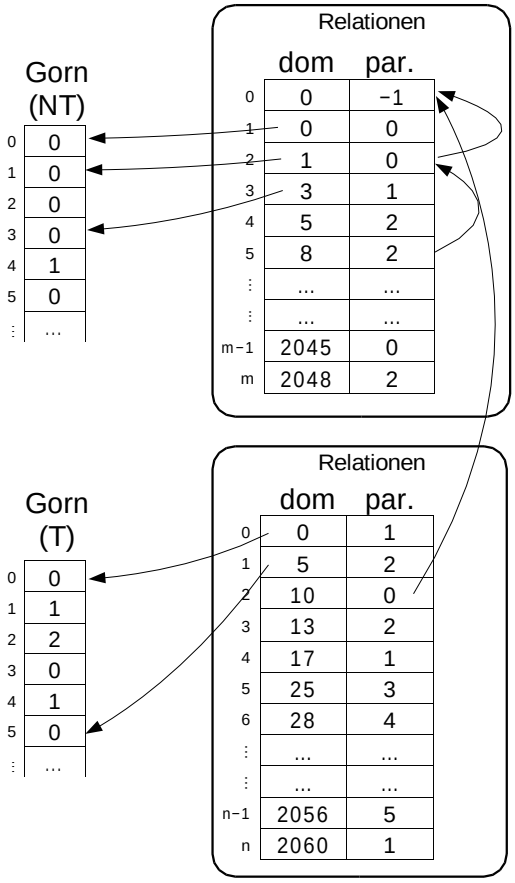


Abbildung 11.7: Kodierung der Gorn-Adressen

Es sei darauf hingewiesen, dass für jeden Knoten eines Graphen zusätzlich zu den Gorn-Adressen noch ein Verweis auf seinen Mutterknoten abgelegt wird. Dieser Verweis wird nicht für die Anfrageauswertung verwendet. Stattdessen

beschleunigt diese zusätzliche Information die Rekonstruktion eines Graphen für die Ergebnisvisualisierung und den Ergebnis-Export (vgl. Teil IV dieser Arbeit).

Als Ansatzpunkte für den Einsatz von Kompressionsverfahren dient zum einen die Feststellung, dass die Verweise in beiden Verweislisten aufsteigend angeordnet sind. Hier kann statt des absoluten Verweiswerts die Differenz zwischen den Zeigerwerten abgelegt werden. Aus der Summe der Differenzen können dann wieder die absoluten Verweise berechnet werden. Vorteil der Differenzenbildung: Gegenüber den absoluten Adressen stellen die Differenzen sehr kleine Werte dar, die durch wenige Bits kodiert werden können. Als weiterer Ansatzpunkt kann für die Buchstaben der Gorn-Adressen festgehalten werden, dass niedrige Nummern dominant sind. Eine weitere Reduzierung des Speicherbedarfs ist hier denkbar.

Zugriffsoperationen

- `byte[] getNTGornAddress(int number)`

  Ermittelt die Gorn-Adresse des nicht-terminalen Knotens mit der Nummer *number*.

- `byte[] getTGornAddress(int number)`

  Ermittelt die Gorn-Adresse des terminalen Knotens mit der Nummer *number*.

- `boolean dominatesN(byte[] v, byte[] w, byte n)`

  Stellt fest, ob die Gorn-Adressen v und w eine Dominanz in n Stufen kodieren.

- ...

  Die Tests für alle übrigen Dominanzrelationen ergeben sich analog.

Ein Dominanzrelations-Test für zwei Graphknoten wird also folgendermaßen durchgeführt: Abhängig von der jeweiligen Knotendomäne werden die Gorn-Adressen der beiden Knoten mit Hilfe der Operationen `getNTGornAddress` und `getTGornAddress` ermittelt. Anschließend kann die spezielle Dominanzrelation mit der dafür vorgesehenen Operation (z.B. `dominatesN`) überprüft werden.

Konstruktion

Nach dem Einlesen aller Knoten eines Graphen wird die Graphstruktur rekonstruiert. Mittels rekursivem Durchlauf wird jedem Knoten seine Gorn-Adresse

sowie sein Elternknoten zugewiesen. Parallel zu den Attributwerten und Knoteneigenschaften wird pro Knoten ein neuer Verweis auf die Gorn-Adresse abgelegt und die Adresse an dieser Stelle der Adressliste eingefügt.

11.3.2 Präzedenz

Datenstruktur

Bei der Repräsentation der Präzedenz-Relation beschränkt sich die Aufgabe auf die Nichtterminal-Knoten, da die Präzedenz der Terminalknoten bereits durch die Knotennummerierung festgehalten wird (vgl. Abschnitt 11.1.2). Für die nicht-terminalen Knoten gibt die eigens auf die vorliegende Datenstruktur angepasste Definition der Präzedenz bereits die naheliegende Repräsentation vor: Sind die linkesten Nachfahren zweier Knoten bekannt, so kann die Präzedenz der Knoten an der Präzedenz der linkesten Nachfahren abgelesen werden.

Deshalb wird für jeden nicht-terminalen Knoten ein Verweis auf seinen linkesten Nachfahren abgelegt (vgl. Abb. 11.8). Der Verweis wird dabei nicht als Verweis auf die absolute Listenpositionsnummer kodiert, sondern auf die relative Knotennummer innerhalb desselben Graphen. Im nachfolgenden Abschnitt wird beschrieben, wie die Graphanfänge und -enden repräsentiert werden, die hier zur Berechnung der Listenpositionsnummer aus der relativen Position benötigt werden.

Da auch der rechteste Nachfahre eine Relation der Korpusbeschreibungssprache darstellt, wird parallel zum linkesten auch der rechteste Nachfahre bestimmt und in einer separaten Liste gespeichert (vgl. Abb. 11.8).

Zugriffsoperationen

- `int getLeftCorner(int position)`

  Bestimmt den linkesten Nachfahren des nicht-terminalen Knotens an der Position *position*. Hier wird die relative Position bzgl. des ersten Knotens desselben Satzes angegeben.

- `int getRightCorner(int position)`

  Bestimmt den rechtesten Nachfahren des nicht-terminalen Knotens an der Position *position*. Hier wird die relative Position bzgl. des ersten Knotens desselben Satzes angegeben.

- `boolean precedesN(int leftcorner1, int leftcorner2, byte n)`

  Überprüft, ob die Nummern der beiden Eckknoten eine Präzedenz mit Abstand n kodieren.

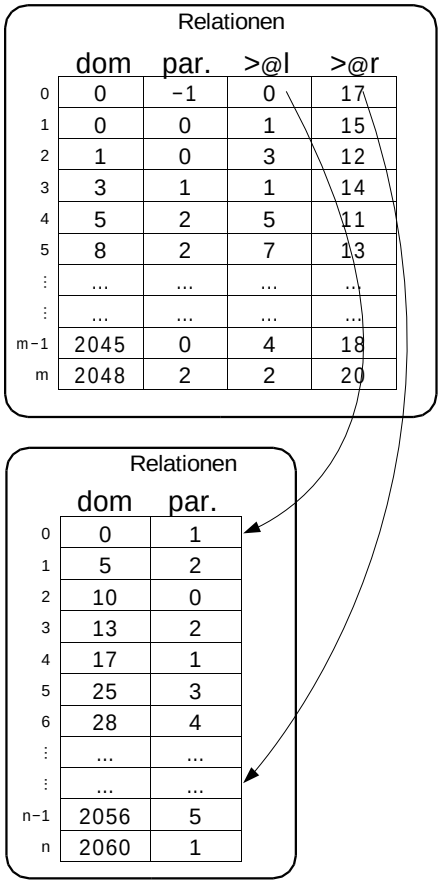


Abbildung 11.8: Präzedenzinformation für nicht-terminale Knoten

- ...

  Die übrigen Präzedenzrelationen werden mit analogen Operation überprüft.

Ein Präzendenzrelations-Test für zwei Graphknoten wird also folgendermaßen durchgeführt: Abhängig von der jeweiligen Knotendomäne wird für einen nicht-terminalen Knoten der linkeste Nachfahre mit Hilfe der Operation `getLeftCornerNode` ermittelt bzw. stellt der terminale Knoten selbst seinen linkesten Nachfahren dar. Anschließend kann die spezielle Präzedenzrelation mit der dafür vorgesehenen Operation (z.B. `precedesN`) überprüft werden.

Der Vollständigkeit halber sei noch bemerkt, dass sich der Test zweier Knoten bzgl. der gemischten Relation `$.*` (Geschwisterrelation mit allgemeiner Präzedenz) aus dem Resultat der beiden speziellen Dominanz- und Präzedenztests zusammensetzt. Zudem sei angemerkt, dass für negierte Relationen keine Indexdaten benötigt werden, da der Test auf eine Relation $v\ !R\ w$ als der negierte Wert der Relation $v\ R\ w$ definiert ist.

Konstruktion

Ermittle für jeden Knoten eines Graphen rekursiv seinen linkesten bzw. rechtesten Nachfahren als den linkesten bzw. rechtesten Knoten unter den linkesten bzw. rechtesten Nachfahren aller seiner Töchter. Der linkeste und rechteste Nachfahre eines Terminalknotens ist der Terminalknoten selbst. Linkester bzw. rechtester Knoten einer Kandidatenmenge ist derjenige Knoten mit der niedrigesten bzw. höchsten Nummerierung.

11.4 Graphen

Mit der Repräsentation der Knotenbeschreibungen, Knoteneigenschaften und Relationen sind die Graphen eines Korpus fast vollständig repräsentiert. Es fehlen lediglich Informationen zur Graphstruktur: *Welche Knoten gehören zu einem vorgegebenen Graphen? Welcher Knoten stellt den Wurzelknoten des Graphen dar?* Die Kodierung dieser Informationen ist im nachfolgenden Unterabschnitt 11.4.1 ausgeführt. Zur vollständigen Beschreibung eines Korpusgraphen fehlen dann lediglich die Graph-ID und die Verwaltung der sekundären Kanten als Besonderheit des vorliegenden Datenmodells. Diese beiden Bestandteile werden in den Unterabschnitten 11.4.2 und 11.4.3 behandelt.

11.4.1 Graphstruktur

Datenstruktur

Da die Attributwerte der Knoten, die Knoteneigenschaften und die Relationsinformationen stets Knoten für Knoten und Graph für Graph abgelegt worden sind, bilden die Nichtterminal- und Terminalknoten eines Graphen einen Nummerierungsblock. Es genügt demnach, einen Zeiger auf die beiden Anfangsknoten eines Graphen zu platzieren (vgl. auch Abb. 11.9).

Die Anzahl der zu einem Graph gehörenden Terminal- bzw. Nichtterminal-Knoten ergibt sich damit aus der Differenz der Anfangspositionen des nächsten und des aktuellen Graphen. Die grau unterlegten Listenzeilen in Abb. 11.9 machen noch einmal optisch deutlich, dass die verschiedenartigen Graphinformationen parallel verwaltet werden.

Neben den Zeigern auf die Graphknoten wird zusätzlich noch ein Verweis auf den Wurzelknoten des Graphen angegeben. Damit steht auch die nötige Information zur Prüfung des `root`-Prädikats zur Verfügung.

Zugriffsoperationen

- `int getNumberOfGraphs()`

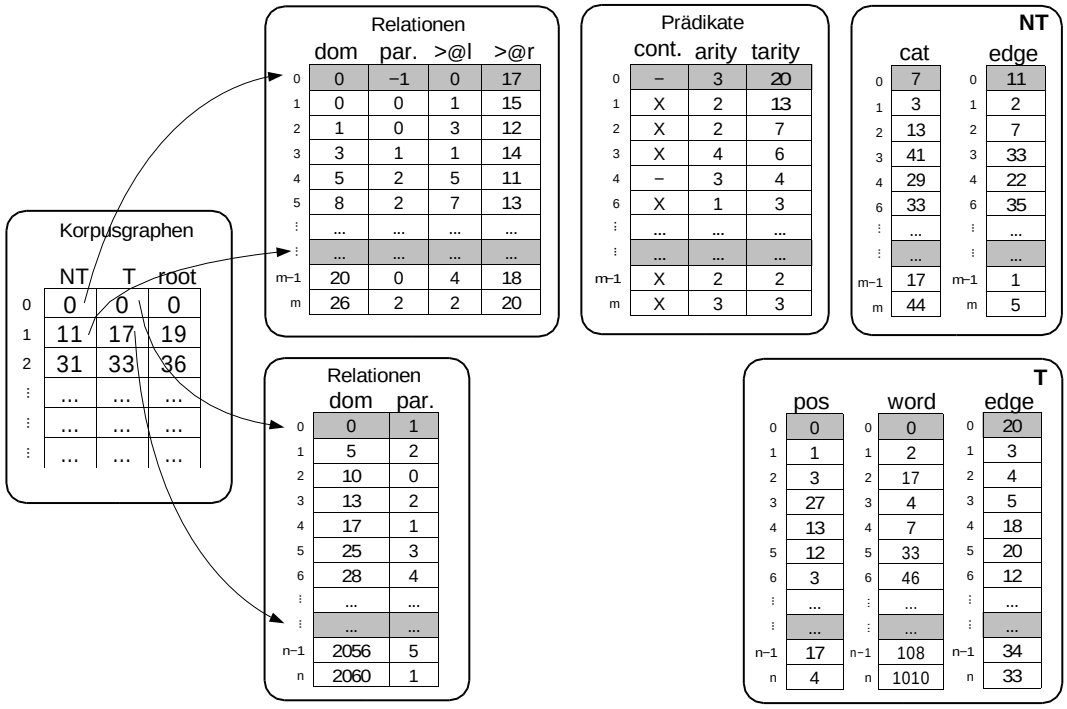


Abbildung 11.9: Blockbildung durch die Knoten eines Satzes

Gibt die Anzahl der Korpusgraphen an.

- `int getFirstNTNode(int graphno)`

  Gibt die Nummer des ersten nicht-terminalen Knotens des Graphen mit der Nummer *graphno* an.

- `int getNumberOfNTNodes(int graphno)`

  Gibt die Anzahl der nicht-terminalen Knoten des Graphen mit der Nummer *graphno* an.

- `int getFirstTNode(int graphno)`

  Gibt die Nummer des ersten terminalen Knotens des Graphen mit der Nummer *graphno* an.

- `int getNumberOfTNodes(int graphno)`

  Gibt die Anzahl der terminalen Knoten des Graphen mit der Nummer *graphno* an.

- `int getRootNode(int graphno)`

Gibt die Nummer des nicht-terminalen Wurzelknotens des Graphen mit der Nummer *graphno* an. Sollte der Wurzelknoten ausnahmsweise ein Terminalknoten sein, d.h. besteht der gesamte Graph nur aus genau einem Terminalknoten, so gibt zeigt die Wurzelkodierung $\Leftrightarrow 1$ diesen Sonderfall an.

Konstruktion

Zähle bei der satzweisen Indexierung stets die Anzahl aller bislang indexierten Terminal- und Nichtterminal-Knoten sowie die Anzahl der jeweiligen Knoten im aktuellen Graph mit. Lege mit Hilfe dieser Informationen die Verweise auf die Graphanfänge fest. Der Wurzelknoten ist in der TIGER-XML-Repräsentation eindeutig gekennzeichnet (vgl. Kapitel 7) und kann dort abgelesen werden.

Anmerkung: Es gibt Baumbanken, deren Korpusgraphen nicht über einen eindeutig bestimmten Wurzelknoten verfügen (z.B. das TIGER-Korpus). In diesem Falle ist der Konstruktionsalgorithmus bewusst robust implementiert worden: Um auch diese Korpora verarbeiten zu können, definiert der Algorithmus einen künstlichen Wurzelknoten, der den gesamten Korpusgraphen umspannt.

11.4.2 Graph-IDs

Datenstruktur

Für die Graph-IDs gelten dieselben Überlegungen wie für die Knoten-IDs. Sie werden lediglich beim Export der Anfrageergebnisse eingesetzt und können daher in einer separaten Textdatei verwaltet werden (vgl. Abb. 11.10).

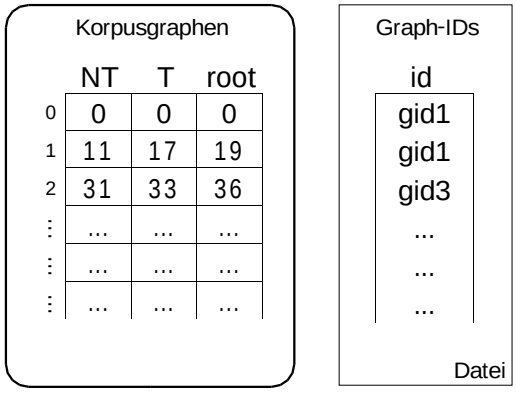


Abbildung 11.10: Repräsentation der IDs der Korpusgraphen

Zugriffsoperationen

- `String getGraphID(int graphno)`

  Ermittelt die ID des Graphen mit der Nummer *graphno*.

Konstruktion

Die Graph-IDs werden graphenweise in die Textdatei geschrieben.

11.4.3 Sekundäre Kanten

Eine Besonderheit des unterstützten Datenmodells stellen die sekundären Kanten dar. Sie können wahlweise über die Beschriftung der sekundären Kante oder unterspezifiziert ohne Beschriftung abgefragt werden. Im Vergleich zu anderen Relationen ist der Gebrauch sekundärer Kanten eher selten. Aus diesem Grunde verwendet der Index eine einfache Lösung zur Repräsentation dieser Kanten.

Datenstruktur

Die sekundären Kanten eines Graphen werden aufgelistet (vgl. Abb. 11.11). Sie sind dabei durch Startknoten, Kantenbeschriftung und Zielknoten eindeutig repräsentiert. Die Knoten werden durch die zugehörigen Knotennummern zusammen mit der Knotendomäne kodiert. Im Beispiel führt vom Terminalknoten mit der Nummer 0 eine beschriftete Kante zum Nichtterminal-Knoten mit der Nummer 7. Die Nummern werden dabei relativ zum ersten Terminal- bzw. Nichtterminalknoten des jeweiligen Graphen angegeben (vgl. Unterabschnitt 11.4.1).

Für die Beschriftung der sekundären Kanten wird analog zu den primären beschrifteten Kanten des Datenmodells ein neues Attribut *secedge* definiert und ein entsprechendes zusätzliches Attributregister angelegt. Damit ergeben sich für ein Korpus bis zu zwei zusätzliche Attribute. Die Kantenbeschriftung kann nun ebenfalls als Nummer angegeben werden, die mit Hilfe des neuen Attributregisters aufgelöst werden kann.

Damit die Zugehörigkeit der sekundären Kanten zu den einzelnen Korpusgraphen feststellbar ist, wird für jeden Graphen ein Verweis auf seine erste sekundäre Kante angegeben. Die Anzahl der zu einem Graph zugehörigen sekundären Kanten ergibt sich dann aus der Differenz des nachfolgenden und des aktuellen Verweises. Sollte es keine sekundäre Kante geben, werden der aktuelle und der nachfolgende Verweis identisch gewählt, woraus sich die Anzahl 0 sekundärer Kanten ergibt (vgl. Graph mit der Nummer 1 in Abb. 11.11).

Durch die aufsteigende Anordnung der Verweise ergibt sich wieder die Möglichkeit der Kompression durch Differenzenkodierung. Dies setzt allerdings eine streng graphenweise Vorgehensweise voraus. Schränkt ein Suchfilter die Graphkandidatenmenge ein, ist diese Art der Komprimierung auch nachteilig, da stets alle Kanteneinträge sequentiell durchlaufen werden müssen.

Zugriffsoperationen

- `int getNumberOfSecondaryEdges(int graphno)`

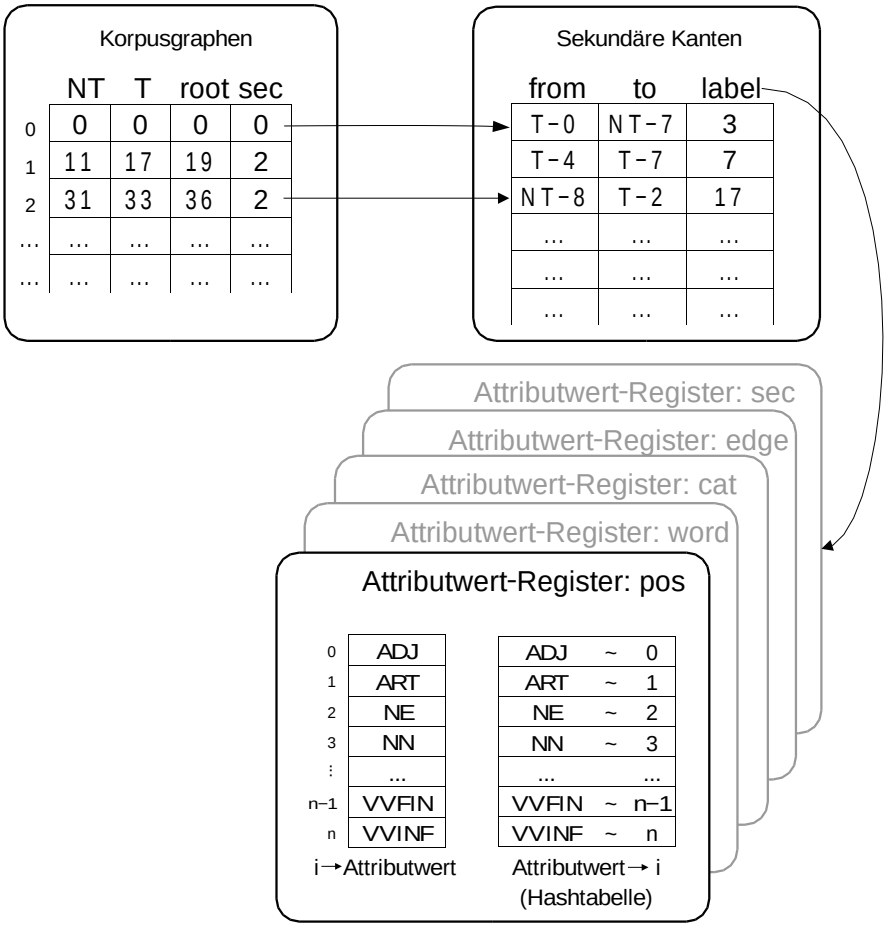


Abbildung 11.11: Verwaltung sekundärer Kanten

Bestimmt die Anzahl der sekundären Kanten des Graphen mit der Nummer *graphno*.

- `int getStartNodeOfSecondaryEdge(int graphno, int secedgeno)`

  Bestimmt die Nummer des Startknotens der sekundären Kante mit der Nummer *secedgeno* des Graphen mit der Nummer *graphno*.

- `int getEdgeLabelOfSecondaryEdge(int graphno, int secedgeno)`

  Bestimmt die Nummer der Kantenbeschriftung der sekundären Kante mit der Nummer *secedgeno* des Graphen mit der Nummer *graphno*.

- `int getTargetNodeOfSecondaryEdge(int graphno, int secedgeno)`

  Bestimmt die Nummer des Zielknotens der sekundären Kante mit der Nummer *secedgeno* des Graphen mit der Nummer *graphno*.

Um Terminal- und Nicht-Terminalknoten bei den beiden Knotenzugriffsoperationen unterscheiden zu können, wird im Falle eines Nichtterminals eine Konstante addiert. Sie ist so groß gewählt, dass sie durch die (relative) Nummer eines Terminalknotens nie erreicht werden kann. Eine mögliche Wahl ist beispielsweise 1000.

Um nun die Relationsbeziehung durch eine sekundäre Kante für zwei Graphknoten v und w testen zu können, werden alle sekundären Kanten des Graphen durchlaufen. Sollte sich darunter eine Kante von v nach w befinden (im Falle von `>~L` mit korrekter Kantenbeschriftung), so ist die Relationsbeziehung erfüllt.

Konstruktion

Die sekundären Kanten eines Korpusgraphen sind unmittelbar an der TigerXML-Korpusrepräsentation abzulesen und werden sequentiell abgelegt.

11.5 Korpus

11.5.1 Korpusdeklaration

Die Indexrepräsentation der Korpusgraphen ist nun vollständig beschrieben. Neben den Graphen müssen die Daten der Korpusdeklaration verwaltet werden. Sie sind zum einen Grundlage für die Anfrageauswertung, da sie die Korpusattribute und ihre Domänen festlegen. Zum anderen umfassen sie auch Meta-Information über das Korpus und können daher bei der Gestaltung der Benutzeroberfläche eingesetzt werden (vgl. Abschnitt 14.2).

Da es sich bei den Daten der Korpusdeklaration fast ausschließlich um elementare Datentypen wie Zeichenkettenlisten o.ä. handelt, werden diese Angaben als Attribute eines dafür eingerichteten Java-Objekts (mit dem Namen *Header*) verwaltet. Die Operationen zum Zugriff auf die Deklarationsdaten werden nun in kompakter Form aufgelistet.

Zugriffsoperationen

- `boolean isTCorpusFeature(String fname)`

- `boolean isNTCorpusFeature(String fname)`

  Prüft, ob der Attributname *fname* im Korpus als Attribut für Terminalknoten bzw. Nicht-Terminalknoten deklariert ist.

- `String[] getTFeatureNames()`
- `String[] getNTFeatureNames()`

  Liefert die Namen aller für Terminalknoten bzw. Nichtterminalknoten deklarierten Attribute.

- `boolean isAttributeValues(String fname)`
- `String[] getAttributeValues(String fname)`

  Stellt fest, ob es zu einem Attribut mit dem Namen *fname* eine Deklaration aller Attributwerte gibt bzw. gibt diese Werte als Liste zurück.

- `boolean isAttributeDescriptions(String fname)`
- `String[] getAttributeDescriptions(String fname)`

  Stellt fest, ob es zu einem Attribut mit dem Namen *fname* eine Beschreibung aller Attributwerte gibt bzw. gibt diese Beschreibungen als Liste zurück.

- `String getCorpusID()`
- `String getCorpusName()`
- ...

  Operationen zur Ermittlung von Meta-Information zum Korpus: ID, Name, Autor usw. (vgl. Abschnitt 5.2).

11.5.2 Typen

Die Typdefinitionen für die definierten Attribute eines Korpus sind in der gegenwärtigen Implementation korpusbezogen gelöst worden. Zwar wäre auch denkbar, dass jeder Anwender seine eigenen Typdefinitionen verwaltet. Doch gestaltet sich die Realisierung einer Client/Server-Architektur (Abschnitt 13.4) in diesem Falle deutlich schwieriger, da der Client mit jeder Anfrage seine aktuell definierten Typdefinitionen an den Server übertragen müsste.

Mit dem Einlesen des Korpus werden also für alle definierten Attribute die Typdefinitionen eingelesen. Die Typdefinitionen sind in separaten Dateien abgelegt. Die Typsymbole einer Typdefinition werden als Bitvektoren repräsentiert. Das i-te Bit kodiert dabei, ob der i-te Attributwert zum Typ gehört oder nicht. Auf diese Weise kann in einem Schritt über die Zugehörigkeit eines Attributwerts entschieden werden. Da Attributwerte in der Typhierarchie genau einmal eingehängt werden dürfen, ergibt sich die Negation eines Typs einfach als negierter Bitvektor. Voraussetzung dafür ist die Definition des Universum, d.h. aller für ein Attribut zulässigen Attributwertkonstanten. Diese werden in der

Korpusdefinition deklariert. Sollte diese Deklaration fehlen, wird ersatzweise die Liste aller verwendeten Attributwerte (vgl. Attributregister) als Universum definiert.

Zugriffsoperationen

Ein Typsystem für ein Attribut wird in einem eigenen Java-Objekt (mit Namen *TypeHierarchy*) verwaltet. Folgende Zugriffsoperationen stehen zur Verfügung:

- `boolean isType(String tname)`

  Prüft, ob ein Typsymbol mit dem angegebenen Namen definiert ist.

- `boolean isConstMemberOfType(String constant, String tname)`

  Prüft, ob die Konstante `constant` im Typ `tname` enthalten ist.

- `boolean isSubType(String tname_a, String tname_b)`

  Prüft, ob der durch `tname_a` bezeichnete Typ im durch `tname_b` bezeichneten Typ enthalten ist.

Die beiden letzen Operationen können mit Hilfe des Bitvektors sehr effizient durchgeführt werden. Für die Auswertung der ersten Operation wird eine Hashtabelle mit den Namen aller Typsymbole herangezogen.

11.6 Zum Stand der Implementation

Alle beschriebenen Bestandteile des Korpusindex sind vollständig implementiert worden. Kompressionstechniken wurden bislang nur an Stellen eingesetzt, an denen besonders hohe Kompressionsraten zu erzielen waren. Der Speicherbedarf des gesamten Index auf der Festplatte (inkl. weiterführender Indexierungsansätze, vgl. Kapitel 13) beträgt überschlagsweise etwa 50 MB pro eine Million Token im Eingangstext. Durch die Ausweitung des Einsatzes von Kompressionstechniken dürfte sich der Speicherbedarf etwa halbieren lassen.

Ein weiteres Manko der aktuellen Implementation besteht darin, dass das Korpus stets vollständig in den Hauptspeicher geladen wird. Die Suche auf sehr großen Korpora ist damit zur Zeit nicht durchführbar. Da die gängigen manuell annotierten Korpora einen Umfang von maximal 2-3 Million Token haben, sind diese jedoch (mit 100-150 MB Hauptspeicherbedarf) auf jedem handelsüblichen PC verarbeitbar.

Zudem ist eine Lösung des mit der Korpusgröße wachsenden Speicherbedarfs bereits in Vorbereitung: Bedingt durch die konsequente graphenweise und knotenweise Anordnung der Daten kann ein Index in große Blöcke eingeteilt werden, deren Daten blockweise eingelesen werden (z.B. Blöcke

mit jeweils 1.000 Korpusgraphen). Bei Erreichen des nächsten Blockes durch den Anfrageverarbeitungs-Prozessor werden die vorausgegangenen Daten dann freigegeben und der nächste Block eingeladen. Damit bleibt der Speicherbedarf konstant und es sind Korpora beliebiger Größe verarbeitbar. Die Wahl des Blockgrößen-Parameters sollte dabei in Abhängigkeit von der vorliegenden Hauptspeichergröße getroffen werden, um möglichst große Korpusblöcke am Stück zu verarbeiten.

Kapitel 12

Anfrageverarbeitung zur Laufzeit

Die Implementation des in Kapitel 9 definierten Kalküls teilt sich in zwei Schritte: Der Index repräsentiert in einem Vorverarbeitungsschritt die Ergebnisse derjenigen Verarbeitungsregeln, die vorab ohne Kenntnis der Anfrage durchgeführt werden können. Es handelt sich dabei insbesondere um die Normalisierungsregeln von Graphbeschreibungen (vgl. Unterabschnitt 9.5.3) und die Verwaltung der Knotenbeschreibungen aller Korpusknoten (vgl. Knotenauswahl-Regel in Unterabschnitt 9.4.3).

Zur Laufzeit werden diese Ergebnisse dann mit der Anfrage verknüpft. Die Implementation dieser zur Laufzeit durchgeführten Anfrageverarbeitung wird im vorliegenden Kapitel beschrieben. Sie wird zunächst in Abschnitt 12.1 grob skizziert. Anschließend werden mit der Anfragenormalisierung, der Repräsentation logischer Variablen und der graphenweisen Anfrageevaluation die zentralen Bausteine des Grobalgorithmus in den Abschnitten 12.2 bis 12.4 detailliert beschrieben. Der abschließende Abschnitt 12.5 geht auf die Korrektheit und Vollständigkeit der Gesamtimplementation ein.

Optimierungen, die für eine korrekte und vollständige Anfrageverarbeitung nicht erforderlich sind, die Verarbeitung aber maßgeblich beschleunigen können, werden im nachfolgenden Kapitel 13 verfolgt: Verwendung von Suchfiltern, Optimierung von Anfragen sowie Java-spezifische Ansätze.

12.1 Grobalgorithmus

Für die Verarbeitung von Anfragen gibt der Kalkül die grundsätzliche Vorgehensweise bereits vor. Um die Ableitbarkeit einer Anfrage `q` aus einem ausgewählten Korpusgraphen `g` festzustellen, muss geprüft werden, ob es eine Knotenauswahl (d.h. eine Zuordnung von Korpusgraphknoten zu Knotenbeschreibungen der Anfrage) gibt, bei der für `g & q` kein Widerspruch $\perp$ abgeleitet werden kann.

Doch im Gegensatz zur Definition des Kalküls ist die Strategie der Auswertung wesentlich für die Effizienz der Implementation. Es geht daher bei der Festlegung des Grobalgorithmus darum, die Verarbeitungsregeln so zu gruppieren und die Verarbeitungsabfolge so zu wählen, dass die enstehende Implementationsstrategie korrekt, vollständig und möglichst effizient arbeitet.

Die nichtdeterministische Auswahl der Korpusgraphen stellt für den Gesamtalgorithmus einen so genannten *choice point* dar. Der Nichtdeterminismus wird zunächst durch eine erschöpfende Auswahl aller Korpusgraphen aufgelöst. Die erschöpfende Vorgehensweise ist dabei natürlich nicht ideal und wird im Rahmen des nachfolgenden Kapitels durch die Entwicklung von Suchraumfiltern verbessert.

Wie bereits aus der Kalküldefinition hervorgeht, besteht die Idee des Verfahrens darin, die Korpusanfrage in eine Disjunktive Normalform zu bringen. Resultat der Disjunktiven Normalform ist eine Liste von Disjunkten, die unabhängig voneinander weiterverarbeitet, d.h. graphenweise gegen das Korpus getestet werden. Die Ergebnisse werden am Ende der Verarbeitung dann wieder zusammengefügt. Vorteil des Normalisierungsansatzes ist die Tatsache, dass auf allen Verarbeitungsebenen (mit Ausnahme der Attributwert-Ebene) konjunktive Kontexte vorliegen. Es müssen keine Variablen in disjunktiven Kontexten verwaltet werden, was die Konzeption des Algorithmus erheblich vereinfacht.

Der Grobalgorithmus (vgl. Abb. 12.1) zeigt die Trennung zwischen der Normalisierungs- und Evaluierungsphase. Der erste Verarbeitungsschritt besteht aus dem Parsen der Anfrage. Es handelt sich hier um die Konvertierung der Anfrage-Zeichenkette in ein Java-Objekt. Dieser Bestandteil der Verarbeitung ist bereits bei der Implementation der Korpusbeschreibungssprache besprochen worden (vgl. Abschnitt 8.3) und wird deshalb hier nicht näher ausgeführt. Obwohl alle weiteren Schritte auf der Java-Objekt-Ebene operieren, werden sie zum besseren Verständnis stets auf textueller Ebene illustriert.

Abschnitt 12.2 wird zeigen, dass die anschließende Normalisierung aus technischen Gründen in die Schritte Pre-Normalisierung und Disjunktive Normalform unterteilt ist. Es entsteht als Ergebnis eine Liste der Disjunkte $q_1, \ldots, q_m$, die nun isoliert voneinander verarbeitet werden. An dieser Stelle ist zu bemerken, dass die isolierte Verarbeitung der Disjunkte natürlich auch durch isolierte Prozesse bzw. Prozessoren erfolgen könnte (vgl. technische Ansätze zur Steigerung der Effizienz in Abschnitt 13.4).

Für die eigentliche Anfrageevaluation im vierten Schritt wird das Haldenmodell verwendet (vgl. Abschnitt 9.1). Die Abschnitte 12.3 und 12.4 beschreiben, wie das Anfragedisjunkt in die Halde eingewiesen (Teilschritt a) und das Korpus graphenweise gegen das Disjunkt getestet wird (Teilschritt b). Mit dem Zusammenbinden der Ergebnisse im Schritt fünf ist der Algorithmus beendet.

1. Parse die Anfrage q.
2. Pre-Normalisiere die Anfrage q.
3. Überführe die Anfrage q in Disjunktive Normalform.
4. Für jedes Disjunkt $q_1, \ldots, q_m$:
   (a) Erzeuge die Halde h zum Disjunkt q_i.
   (b) Für jeden Korpusgraphen $g_1, \ldots, g_n$:
   - Teste das Disjunkt q_i mit der Halde h auf dem Korpusgraphen g_j.
5. Füge die Anfrageergebnisse der Disjunkte zusammen.

Abbildung 12.1: Verarbeitung von Korpusanfragen - Grobalgorithmus

12.2 Anfragenormalisierung

Die Aufgabe der Anfragenormalisierung besteht darin, die Anfrage in eine Disjunktive Normalform zu überführen. Einige Vorverarbeitungsschritte werden dabei explizit vorgezogen (vgl. Unterabschnitt 12.2.1), bevor die eigentliche Normalisierung erfolgt (vgl. Unterabschnitt 12.2.2).

12.2.1 Pre-Normalisierung

Der erste Schritt der Pre-Normalisierung besteht aus der Vereinheitlichung der Knotenbeschreibungen und Attributspezifikationen als Vorbereitung für die Umformung in die Disjunktive Normalform. Alle Knotenbeschreibungen und Attributspezifikationen, die bislang nicht an eine Variable der entsprechenden Stufe gebunden sind, werden hier mit einer ungebundene Variable verknüpft (vgl. DNF-Vorbereitungen von Knotenbeschreibungen in Unterabschnitt 9.4.1). Die beiden Randfälle `[]` und `#x:[]` werden auf `#x` abgebildet (vgl. Unterabschnitt 9.4.2). Beispiel:

`[pos="NN" & word="Haus"]` $\Rightarrow$ `#v:[#f:pos="NN" & word="Haus"]`

Der zweite Schritt ist kein expliziter Verarbeitungsschritt des Kalküls. Seine Aufgabe besteht darin, den späteren Test von Relationen und Prädikaten zu vereinfachen, indem zu testende Knotenbeschreibungen stets zuvor gebunden werden. Im folgenden Beispiel müsste ein Relationstest vor dem eigentlichen Test zunächst den Knoten `#v` binden:

```
#v:[pos="NN" & word="Haus"] > #w
```

Wird die Anfrage hingegen leicht umstrukturiert, so ist die Knotenrelation frei von Bindungsaufgaben:

```
⇒ #v:[pos="NN" & word="Haus"] & #v > #w
```

Auf diese Weise werden die Aufgaben der Anfragebestandteile aufgetrennt. Dadurch wird die Implementation der Tests und Bindungen transparenter und übersichtlicher.

Für diese vereinfachende Umformung müssen zwei Fälle berücksichtigt werden. Ist eine Knotenbeschreibung innerhalb einer Relation durch eine Attributspezifikation ergänzt, so wird diese Spezifikation wie im oben angegebenen Beispiel in Form einer neuen Knotenbeschreibung vorgezogen:

```
#v1:[#f1:cat="NP"] > #v2:[#f2:pos="NN"] ⇒
#v1:[#f1:cat="NP"] & #v2:[#f2:pos="NN"] & #v1 > #v2
```

Für ein Prädikat sieht die Umformung folgendermaßen aus:

```
arity(#v1,3) ⇒ #v1 & arity(#v1,3)
```

Es sei der Vollständigkeit halber angemerkt, dass die Semantik der Anfrage durch diese Umformungen nicht verändert wird.

12.2.2 Disjunktive Normalform

Die Berechnung der Disjunktiven Normalform gliedert sich in zwei Schritte. Im ersten Schritt werden alle Negationen so weit wie möglich nach innen geschoben. Als Ergebnis dieser Verarbeitung können Negationen nur noch unmittelbar vor Konstanten oder Typen stehen. Im zweiten Schritt wird die entstandene Negations-Normalform zu einer Disjunktiven Normalform ausgebaut.

Negations-Normalform

Da Negationen in der Korpusbeschreibungssprache nur bis hin zur Attributspezifikations-Ebene definiert sind, können sie blockweise von außen nach innen gedrückt werden. Die Herstellung der Negations-Normalform gliedert sich in zwei Schritte:

- 1. Schritt: Attributspezifikationen (Verarbeitungsblock: Attributspezifikationen - Disjunktive Normalform)

  Solange noch eine Normalisierungsregel für Attributspezifikationen anwendbar ist (mit Ausnahme der Distributionsregel), wende diese Regel an (vgl. DNF-Regeln in Unterabschnitt 9.3.1 und die Negationsnormalisierungsregel in Unterabschnitt 9.3.3). Beispiele:

  ```
  pos != "NN"  ⇒  pos = !"NN"
  !(pos="NN")  ⇒  NT | pos = !"NN"
  ```

Das zweite Beispiel ist insofern interessant, als die Semantik der Negation auch ausdrückt, dass die Domäne des Attributs nicht gültig sein könnte. Für die Negation des terminalen Attributs `pos` (d.h. Wortart) heißt dies, dass es sich hier auch um ein Nichtterminal handeln könnte.

- 2. Schritt: Attributwerte (Verarbeitungsblock: Attributwerte - Disjunktive Normalform)

  Solange noch eine Normalisierungsregel für Attributwerte anwendbar ist (mit Ausnahme der Distributionsregel), wende diese an (vgl. DNF-Regeln in Unterabschnitt 9.2.1). Beispiel:

  ```
  pos = !("NN"|"NE")  ⇒  pos = (!"NN" & !"NE")
  ```

Disjunktive Normalform

Nachdem Negationen in der Negations-Normalform nur noch unmittelbar vor Konstanten oder Typen stehen können, besteht die Aufgabe der folgenden Verarbeitung darin, die Distributionsregeln des Kalküls solange anzuwenden, bis alle Verarbeitungsebenen (mit Ausnahme der Attributwert-Ebene) disjunktfrei sind.

Bei der Berechnung der Disjunktiven Normalform ist zu beachten, dass die vier Normalisierungsebenen Attributwert, Attributspezifikation, Knotenbeschreibung und Graphbeschreibung nicht isoliert voneinander betrachtet werden können. Ursache dafür ist die Verarbeitungsregel auf Knotenbeschreibungsebene, die eine Knotenbeschreibung in zwei Beschreibungen aufteilt (vgl. Unterabschnitt 9.4.1):

$$\#x_1 : [\ \#x_2 : (k_1\ |\ k_2)\ ] \quad \Longrightarrow \quad \#x_1 : [\ \#x_2 : k_1\ ]\ |\ \#x_1 : [\ \#x_2 : k_2\ ]$$

Zu Gunsten einer möglichst systematischen und effizienten Vorgehensweise wird daher zunächst eine Disjunktive Normalform auf Attributspezifikationsebene hergestellt, bevor die weiteren Ebenen von innen nach außen weiterverarbeitet werden. Die Normalisierung gliedert sich damit in vier Schritte:

- 1. Schritt: Attributspezifikationen (Verarbeitungsblöcke: Attributspezifikationen - Distributionsregel; Attributspezifikationen - Disjunktionsreduktionsregeln)

  Solange die Distributionsregel für Attributspezifikationen anwendbar ist, wende diese Regel an (vgl. Unterabschnitt 9.3.1). Wende auch Disjunktionsreduktionsregeln an (vgl. Unterabschnitt 9.2.2). Dadurch werden unnötige Disjunkte vermieden und die spätere Anzahl der Graphbeschreibungsdisjunktionen reduziert. Beispiel:

  ```
    (pos="NN") & (pos="NE" | word="Haus")  ⇒
  (pos="NN" & pos="NE") | (pos="NN" & word="Haus")
  ```

Ein wichtiger Bestandteil der Normalisierung ist die Ersetzung eines regulären Ausdrucks durch die Disjunktion aller Attributwerte, die diesem Ausdruck entsprechen. Dazu wird die Indexoperation `String[] getFeatureValues()` des entsprechenden Attributregisters verwendet (vgl. Abschnitt 11.1.1), um eine Liste aller Attributwerte des vorliegenden Attributes zu generieren. Diese Liste wird nun durchwandert und alle diejenigen Attributwerte als Konstanten in die Disjunktion übernommen, die den regulären Ausdruck matchen. Beispiel:

```
word = /.*chen/  ⇒  word = ("Häuschen"|"Tischchen")
```

Im Falle eines negierten regulären Ausdrucks ensteht entsprechend als Ergebnis eine konjunktiv verknüpfte Liste von negierten Konstanten.

- 2. Schritt: Attributwerte (Verarbeitungsblöcke: Attributwerte - Distributionsregel; Attributwerte - Disjunktionsreduktion)

  Solange die Distributionsregel für Attributwerte anwendbar ist, wende diese an (vgl. DNF-Regeln in Unterabschnitt 9.2.1). Beispiel:

```
pos = "NN" & ("NN"|"NE")  ⇒
pos = ("NN"&"NN") | ("NN"&"NE")
```

  Versuche auch, Distributionsreduktionen zur Vereinfachung der Attributwerte durchzuführen.

- 3. Schritt: Knotenbeschreibungen (Regel: Knotenbeschreibungen - Disjunktion von Attributspezifikationen)

  Wende soweit möglich die Distributionsregel für Disjunktionen von Attributspezifikationen innerhalb von Knotenbeschreibungen an (vgl. Unterabschnitt 9.4.1). Dadurch entstehen ggf. neue Knotenbeschreibungen. Beispiel:

```
#v:[#f: pos="NN" | word="Haus"]  ⇒
#v:[#f: pos="NN"] | #v:[#f: word="Haus"]
```

- 4. Schritt: Graphbeschreibungen (Regel: Graphbeschreibungen - Distributionsregel)

  Wende soweit möglich die Distributionsregel für Graphbeschreibungen an (vgl. Unterabschnitt 9.5.1). Beispiel:

```
(#v:[#f: pos="NN"] | #v:[#f: word="Haus"]) &
(#np:[#g: cat="NP"] & #np>#v)
```

$\Rightarrow$

```
(#v:[#f: pos="NN"] & #np:[#g: cat="NP"] & #np>#v)  |
(#v:[#f: word="Haus"]) & #np:[#g: cat="NP"] & #np>#v)
```

Die resultierende Formel besteht nun aus einer Disjunktion $g_1|\ldots|g_n$ von Graphbeschreibungen, die anschließend isoliert voneinander weiterverarbeitet werden. Da Disjunktionen innerhalb von Attributspezifikationen nach außen und Negationen nach innen normalisiert worden sind, stehen Attributspezifikationen stets in einem rein konjunktiven Kontext.

Attributwerte hingegen können durchaus Disjunktionen umfassen, allerdings sind die Werte als Disjunktive Normalform angeordnet. In der Praxis liegt in den meisten Fällen eine Disjunktion von Literalen vor (d.h. von Konstanten, Typen oder negierten Konstanten und Typen), doch sind auch echte Konjunktionen möglich (z.B. `pos=(!"NN"&!"NE")`). Disjunktionen auf Attributwert-Ebene werden absichtlich nicht nach außen getragen, um nicht zu viele Graphbeschreibungs-Disjunkte als Ergebnis der Normalisierung zu erhalten.

12.3 Repräsentation logischer Variablen

Wie bereits bei der Kalküldefinition festgelegt worden ist (vgl. Abschnitt 9.1), werden die logischen Variablen durch ein Haldenmodell realisiert. Die Halde hat die Aufgabe, die Repräsentation der Variablenbindungen zu übernehmen. Dieser Abschnitt befasst sich mit dem Aufbau der Halde zu einem vorgegebenen Anfragedisjunkt. Dieser Vorgang stellt die unmittelbare Vorbereitung für den graphenweisen Test des Disjunkts dar. Im ersten Unterabschnitt 12.3.1 wird die zur Repräsentation des Haldenmodells verwendete Datenstruktur beschrieben. Die folgenden beiden Unterabschnitte 12.3.2 und 12.3.3 zeigen auf, wie die zum Disjunkt gehörende Halde erzeugt wird und wie ein erster Konsistenzcheck eine evtl. bereits in der Anfrage vorhandene Inkonsistenz aufdeckt.

12.3.1 Umsetzung des Haldenmodells

Gemäß der Kalküldefinition besteht die Halde aus einer Constraint- und einer Variablenumbenennungs-Halde (vgl. Abschnitt 9.1). In der Constraint-Halde werden Bedingungen an Attributwerte, Attributspezifikationen und Knotenbeschreibungen verwaltet. Bedingt durch die unterschiedliche Struktur dieser Bedingungsebenen teilt sich die Halde in der Implementation in insgesamt vier Teile, die im Folgenden beschrieben werden. Zur Illustration wird eine Beispielhalde mit den vier Teilhalden verwendet. Die Teilhalden werden als Tabellen modelliert, in denen Variablen ihre Bedingungen zugeordnet werden.

1. Knotenbeschreibungen

Knotenvar.	Attr.-Spez.	Knoten-ID
#v1	#f1	2 (NT)
#v2	#f2	–

Für jede Knotenbeschreibungs-Variable wird zum einen festgehalten, an welche Attributspezifikations-Variable sie gebunden ist. Hier kommt der Pre-Normalisierungsschritt zum Tragen, der sichergestellt hat, dass alle Attributspezifikationen mit einer entsprechenden Variablen ausgestattet sind. Im obigen Beispiel findet sich die Bindung der Knotenbeschreibungsvariablen `#v1:[#f1:cat="NP"]` in den ersten beiden Spalten der ersten Zeile wieder.

Als zweite Information wird notiert, ob die Knotenbeschreibungsvariable bereits an einen Knoten gebunden ist, d.h. ob sie bereits mit einer Knoten-ID eines Korpusknotens identifiziert wurde. Zu Beginn der Anfrageauswertung ist noch keine Knotenbeschreibungsvariable an eine Knoten-ID gebunden. Die Knoten-ID wird in die dritte Spalte der Tabelle eingetragen.

Wie bereits bei der Vorstellung des Index angedeutet, werden statt der ursprünglichen Knoten-IDs die Knotennummern als Ersatz-IDs verwendet (vgl. Abschnitt 11.1.4). So ist in der ersten Zeile die Knotenvariable `#v1` an den Nichtterminal-Knoten mit der Nummer 2 gebunden, die Knotenvariable `#v2` in der zweiten Zeile ist dagegen ungebunden. Die Information, dass es sich um ein Nichtterminal handelt (vgl. erste Zeile), dient hier nur zur besseren Lesbarkeit. Wie der nächste Abschnitt zeigen wird, ist sie aus der Tabelle für Attributspezifikationen ablesbar.

2. Attributspezifikationen

Attr.-Spez.-Var.	Typ	Attributspezifikation
#f1	NT	cat=#d1
#f2	T	pos=#d2 & word=#d3

Einer Attributspezifikationsvariablen wird in dieser Tabelle erwartungsgemäß ihre Spezifikation zugeordnet. Doch teilt sich die Spezifikation hier in zwei Teile. Die auf dieser Ebene zulässigen Typen (`T`, `NT`, `FREC`) werden gesondert in einer eigenen Spalte verwaltet. Diese Trennung hat große Vorteile, wenn es um die Implementation der Konsistenzprüfungen geht (vgl. Unterabschnitt 12.3.3). Aber auch die Domäne einer bereits gebunden Knotenbeschreibungsvariablen lässt sich so unmittelbar am Typ der zugehörigen Attributspezifikation ablesen (s.o.).

Es sei an dieser Stelle bemerkt, dass die Erzeugung der Disjunktiven Normalform dazu führt, dass Attributspezifikationen disjunktfrei sind.

Die konjunktiv verknüpften Attribut-Wert-Paare einer Attributspezifikation können daher als Liste verwaltet werden. Zudem werden die Verarbeitungsregeln zur Erzeugung der Halde dazu führen, dass die Werte der Attribut-Wert-Paare lediglich Attributwert-Variablen umfassen werden (Beispiel: `cat=#d1`). Die Bedingungen an diese Variablen werden in der Halde auf der nun folgenden Ebene der Attributwerte verwaltet.

3. Attributwerte

Attr.-Wert-Var.	Attributwert
#d1	"NP"
#d2	"NN"
#d3	"Haus" | "Maus"

Einer Attributwert-Variablen wird auf dieser Ebene ein Attributwert zugeordnet. Es ist zu beachten, dass Attributwerte nicht unbedingt Konstanten sein müssen (vgl. dritte Zeile in der Beispieltabelle). Durch die Normalisierungsphase ist allerdings gesichert, dass sie als Disjunktive Normalform strukturiert sind. Ein Attributwert wird daher in der Implementation als Liste von Disjunkten verwaltet.

4. Umbenennungshalde

Variable	Variable
#d4	#d1
#d5	#d1
#f3	#f2

Es verbleibt die Verwaltung von Variablenumbenennungen. Hier wird einem Variablennamen in Form einer Hashtabelle derjenige Name zugeordnet, auf den er abgebildet worden ist. Die Tabelle stellt also die Implementation der Funktion `deref` des Kalküls dar (vgl. Abschnitt 9.1).

12.3.2 Erzeugen der Halde

Bevor die eigentliche Anfrageverarbeitung gestartet werden kann, muss die Halde zur gegebenen Anfrage bzw. zum aktuellen Anfragedisjunkt erzeugt werden. Diese Erzeugung wird durch die Kalkülverarbeitungsregeln der Attributnormalisierung (vgl. Unterabschnitt 9.3.3; obere drei Regeln) und die Normalisierung von Knotenbeschreibungen (vgl. Unterabschnitt 9.4.2; erste Regel nur für Korpusknoten anwendbar) definiert.

Die Aufgabe des Aufbaus der Halde besteht darin, alle Bedingungen innerhalb von Knotenbeschreibungen in die Halde einzuweisen. Die Anfrage besteht am Ende nur noch aus Knotenbeschreibungen sowie Knotenrelationen und Prädikaten, deren Knotenbeschreibungs-Bestandteile Variablen sind. Die ursprüng-

liche Anfrage ist damit in eine Art Graphbeschreibungs-Skelett und die Halde aufgeteilt.

Der folgende Algorithmus realisiert die Einweisung der Bedingungen in die Halde. Das nachfolgende Beispiel illustriert diesen Vorgang und den Teilungseffekt des Haldenaufbaus.

Der Algorithmus

Der Algorithmus wird für jede Knotenbeschreibung innerhalb des Anfragedisjunkts aufgerufen. Es sei angemerkt, dass der Anfrage-Parser bereits Typkonflikte von Variablen im Hinblick auf eingebaute Typen des Kalküls aufdeckt (vgl. Unterabschnitt 5.3.1). D.h. jeder Variablenname wird innerhalb einer Anfrage in genau einem der folgenden Kontexte verwendet: Knotenbeschreibungen, Attributspezifikationen, Attributwert desselben Attributs. Mögliche Konflikte wie im Falle von `[#x:pos=#y] & [pos=#x]` müssen daher hier nicht in Betracht gezogen werden.

1. Knotenbeschreibungen (Verarbeitungsblock: Knotenbeschreibungen - Normalisierung von Knotenbeschreibungen)

   Verzweige in Abhängigkeit von der Art der Knotenbeschreibung:

   (a) `#x1:[#x2]`

   Weise das Paar $\langle \#x1, \langle \top, \#x2 \rangle \rangle$ in die Knotenbeschreibungstabelle ein, d.h. weise der Variablen `#x1` die Attributspezifikation `#x2` zu und deklariere `#x1` als nicht an eine Knoten-ID gebunden:

Knotenvar.	Attr.-Spez.	Knoten-ID
#x1	#x2	–

   Ist `#x1` bereits an eine andere Attributspezifikation `#x3` gebunden, so werden `#x2` und `#x3` unifiziert. D.h. `#x3` wird durch einen Eintrag in der Variablenumbenennungs-Halde auf `#x2` abgebildet und die an `#x2` und `#x3` gebundenen Attributspezifikationen werden konjunktiv verknüpft (vgl. vorletzte Regel im Verarbeitungsblock). Beispielanfrage: `#v:[#f:cat="NP"] & #v:[#g:cat="VP"]`.

   (b) `#x1:[#x2:k]`

   Weise $\langle \#x1, \langle \top, \#x2 \rangle \rangle$ in die Knotenbeschreibungstabelle ein (Fall a). Weise `#x2:k` in die Attributspezifikations-Tabelle ein (vgl. Schritt 2).

   (c) `#x1`

   Erzeuge eine freie Attributspezifikationsvariable `#x2` und weise `#x1:[#x2]` in die Halde ein (Fall a).

Bemerkungen: Die beiden Formen der Knotenbeschreibung `[]` und `#x:[]` sind bereits bei der Pre-Normalisierung auf `#x` abgebildet worden. Die beiden Regeln in Unterabschnitt 9.4.2 sind also hier nicht mehr anwendbar. Die erste Regel des Verarbeitungsblocks Normalisierung ist hier nicht anwendbar, sie bezieht sich auf die Einweisung eines Korpusknotens in die Halde.

2. Attributspezifikationen (Verarbeitungsblock: Attributspezifikationen - Attributnormalisierung - Regeln 1 bis 3)

   Eine Attributspezifikation `#x2:k` besteht durch die Normalisierung aus der konjunktiven Aufzählung von Attribut-Wert-Paaren und Attributspezifikations-Typen. Die Konjunkte werden nun der Reihe nach (d.h. in der Reihenfolge ihrer Definition in der Anfrage) und in Abhängigkeit von ihrer Form verarbeitet.

   Falls es bereits einen Eintrag für `#x2` gibt, so wird die Attributspezifikation `#x2` in der Anfrage mehr als einmal spezifiziert. Beispiel: `[#f:pos="NN"] & [#f:word="Haus"]` Die Spezifikationen werden in diesem Falle unifiziert (vgl. Unterabschnitt 5.4.3). Dazu werden die Typen und Attribut-Wert-Paare konjunktiv mit den bisherigen Einträgen verknüpft. Da beide Spalteneinträge lediglich aus Konjunktionen bestehen, ändert sich durch Hinzufügen weiterer Konjunkte der konjunktive Kontext nicht.

   Verzweige in Abhängigkeit von der Form des Konjunkts:

   (a) `f=#x3:d`

   Weise den Attributwert `#x3:d` in die Attributwert-Tabelle ein (siehe Schritt 3). Ersetze in der Attributspezifikation den Ausdruck `f=#x3:d` durch `f=#x3`. Füge die Domäne des Attributs `f`, d.h. `T`, `NT` oder `FREC` in die Typspalte ein. Falls dort bereits ein Eintrag vorhanden ist, verknüpfe die beiden Formeln konjunktiv. In der folgenden Beispiel-Tabelle ist `T` als Domäne von `f` angenommen:

Attr.-Spez.-Var.	Typ	Attributspezifikation
`#x2`	`T`	`f=#x3`

   (b) `f=d`

   Ergänze den Ausdruck durch eine ungebundene Variable `#x3` und wende den Fall a) auf den resultierenden Ausdruck `f=#x3:d` an.

   (c) `f=#x3`

   Füge die Domäne des Attributs `f` (`T`, `NT` bzw. `FREC`) in die Typspalte ein. Falls dort bereits ein Eintrag vorhanden ist, verknüpfe die beiden Formeln konjunktiv. Füge `#x3` als ungebundene Attributwert-Variable in die Attributwerte-Tabelle ein.

Man beachte, dass die zugehörige Kalkülregel (vgl. Unterabschnitt 9.3.3) vereinfacht angewendet werden kann, da Typkollisionen wie `pos=#x & word=#x` durch den Anfrage-Parser abgefangen werden.

(d) `t`
Füge `t` in die Typspalte ein. Falls dort bereits ein Eintrag vorhanden ist, verknüpfe die beiden Formeln konjunktiv.

3. Attributwerte (Verarbeitungsblock: Attributspezifikationen - Regel 3)

   Für Attributwerte gibt es nur den Fall `#x3:d` (im Falle von `#x3` füge `#x3:Constant` ein). Der Variablenname und der Attributwert werden in die entsprechenden Spalten eingewiesen:

Attr.-Wert-Var.	Attributwert
#x3	d

   Falls es für eine Variable `#x3` bereits einen Eintrag `c` gibt, so ist der Attributwert mehr als einmal in der Anfrage spezifiziert. Beispiel: `[case=(#c:"nom"|"akk")] . [case=#c:"nom"]`. In diesem Falle findet eine Unifikation statt (vgl. Unterabschnitt 5.4.2). Dazu werden `c & d` konjunktiv verknüpft. An dieser Stelle muss der Algorithmus zur Berechnung der Disjunktiven Normalform auf Attributwert-Ebene angewendet werden (vgl. Unterabschnitt 12.2.2).

Laufzeitanalyse: Der Aufwand zur Erzeugung der Halde wächst linear mit der Anzahl der Formelbestandteile. Er ist damit als sekundär einzustufen. Im Falle einer Unifikation von Attributwerten (vgl. letzten Absatz) kommt der Aufwand zur Erzeugung der Disjunktiven Normalform hinzu, doch sind diese Fälle eher selten.

Beispiel

Die folgende Beispielanfrage beschreibt eine Nominalphrase, die ein Adjektiv und ein Nomen umfasst, die bzgl. des Kasus übereinstimmen. Diese Anfrage ist bereits in Disjunktiver Normalform. Sie stellt das einzige Disjunkt der Anfrage dar.

```
#np:[#f1: cat="NP"] &
#t1:[#f2: pos="ADJA" & case=(#c:"NOM"|"AKK")] &
#t2:[#f3: pos="NN" & case=#c] &
#np > #t1 & #np > #t2 & #t1 . #t2
```

Durch die Anwendung des beschriebenen Algorithmus wird die folgende Halde erzeugt. Man beachte, dass sich die Kasus-Kongruenz der Anfrage in der Halde durch die Bindung an den gemeinsamen Attributwert `#c` ausdrückt.

- Knotenbeschreibungen

Knotenvar.	Attr.-Spez.	Knoten-ID
#np	#f1	–
#t1	#f2	–
#t2	#f3	–

- Attributspezifikationen

Attr.-Spez.-Var.	Typ	Attributspezifikation
#f1	NT	cat=#d1
#f2	T	pos=#d2 & case=#c
#f3	T	pos=#d3 & case=#c

- Attributwerte

Attr.-Wert-Var.	Attributwert
#d1	"NP"
#d2	"ADJA"
#d3	"NN"
#c	"NOM" | "AKK"

- Umbenennungshalde

  Diese Halde ist am Ende der Haldenerzeugung leer.

Das Anfragedisjunkt besteht nach der Haldenerzeugung nur noch aus einer skelettartigen Graphbeschreibung:

```
#np & #t1 & #t2 &
#np > #t1 & #np > #t2 & #t1 . #t2
```

12.3.3 Konsistenzcheck

Nach der Aufteilung der Anfrage in das Anfrageskelett und die Halde wird ein Konsistenzcheck durchgeführt. Der Konsistenzcheck wird hauptsächlich im Rahmen des Tests eines Korpusgraphen gegen die Anfrage gebraucht, um festzustellen, ob nach einer Knotenverschmelzung ein Clash $\perp$ abgeleitet werden kann oder nicht (vgl. Unterabschnitt 12.4.2).

Doch manche Anfragen sind bereits für sich inkonsistent, d.h. sie sind für keinen Korpusgraphen herleitbar. Solche Inkonsistenzen entstehen meist durch Denk- oder Tippfehler des Anwenders. Beispiele:

a) `[pos = ("NN"&"NE")]`
b) `[cat="NP" & pos="NN]`
c) `[pos=#c:"NN"] . [pos=#c:"NE"]`

Dabei ist zu beachten, dass eine Anfrage nur dann inkonsistent ist, wenn alle ihre Disjunkte inkonsistent sind. Sollte nur ein Teil der Disjunkte inkonsistent sein, ist lediglich die Evaluation dieser Disjunkte überflüssig (vgl. Disjunktionsreduktions-Regel für Graphbeschreibungen in Unterabschnitt 9.5.2). Nur wenn alle Disjunkte inkonsistent sind, ist die Auswertung der Anfrage abzubrechen und dem Anwender eine entsprechende Mitteilung zu präsentieren.

Um die weitere Anfrageverarbeitung zu vereinfachen, werden parallel zu den Konsistenzchecks auch Disjunktionsreduktionen auf Attributwert-Ebene durchgeführt. Die Attributspezifikationsebene ist hingegen nach der Normalisierung disjunktfrei.

Der Konsistenzcheck ist in zwei Bereiche unterteilt. Die Hauptquelle für Inkonsistenzen liegt in der Halde. Hier werden alle Attributwerte (vgl. Unterabschnitt 9.2.2 und Unterabschnitt 9.2.3) und Attributspezifikationen (vgl. Unterabschnitt 9.3.4) geprüft. Einträge in der Knotenbeschreibungs-Tabelle können nur dann inkonsistent werden, wenn sie an zwei verschiedene Knoten-IDs gebunden werden sollen.

Inkonsistenzen der Halde werden auf die Graphbeschreibungsebene weitergereicht. Auf diese Weise werden Konsistenzen von innen nach außen getragen, d.h. von der Attribut-Wert-Ebene bis auf die Graphbeschreibungsebene. Zudem kann eine Graphbeschreibung inkonsistent sein, falls eine Knotenrelation `#v R #v` verwendet wird. Denn alle definierten Relationen sind nicht-reflexiv.

Der implementierte Konsistenzcheck-Algorithmus arbeitet von innen nach außen. Seine Aufgabe besteht darin, eine evtl. Inkonsistenz aufzudecken und an die nächste Ebene weiterzureichen. Da zu diesem Zeitpunkt bereits auf der Disjunktiven Normalform und damit auf rein konjunktiven Kontexten gearbeitet wird, gelangt jede Inkonsistenz `#d`$:\bot$ auf Attributwerte-Ebene bzw. `#f`$:\bot$ auf Attributspezifikations-Ebene ganz nach außen (vgl. Propagierungsregeln auf den einzelnen Ebenen). D.h. bei Entdecken einer Inkonsistenz kann die Verarbeitung sofort abgebrochen werden. Der Algorithmus wird nun detailliert beschrieben.

Der Algorithmus

1. Attributwerte (Verarbeitungsblöcke: Attributwerte - Konsistenzcheck; Attributwerte - Disjunktionsreduktion)

   Unmittelbar nach der Erzeugung der Halde befinden sich alle Attributwerte in Disjunktiver Normalform, d.h. sie sind von der folgenden Form ($l_{i,j}$ sind Literale):

   $$(l_{1,1} \,\&\, \ldots \,\&\, l_{1,n_1}) \mid \cdots \mid (l_{m,1} \,\&\, \ldots \,\&\, l_{m,n_m})$$

Werden später Anfrage und Korpusgraph verknüpft, so kann lediglich im Rahmen eines Knotentests eine Konstante c konjunktiv mit der oben dargestellten Form verknüpft werden (vgl. Unterabschnitt 12.4.2):

$$((l_{1,1} \,\&\, \ldots \,\&\, l_{1,n_1}) \mid \cdots \mid (l_{m,1} \,\&\, \ldots \,\&\, l_{m,n_m})) \,\&\, c$$

Doch kann dieser Ausdruck durch wenige Umformungen leicht wieder in Disjunktive Normalform gebracht werden:

$$(l_{1,1} \,\&\, \ldots \,\&\, l_{1,n_1} \,\&\, c) \mid \cdots \mid (l_{m,1} \,\&\, \ldots \,\&\, l_{m,n_m} \,\&\, c)$$

Für die folgenden Verarbeitungen kann also in beiden Fällen eine Disjunktive Normalform vorausgesetzt werden.

(a) Konsistenzcheck

Untersuche nacheinander alle Disjunkte. Durch die Normalisierung stellt jedes Disjunkt eine Konjunktion von Literalen dar, d.h. eine Konjunktion von Konstanten, Typen, negierten Konstanten und negierten Typen.

Bilde alle Paare von Literalen und versuche, jedes Paar durch die Anwendung der Konsistenzcheck-Regeln für Attributwerte zusammenzufassen (vgl. Unterabschnitt 9.2.3). Bei erfolgreicher Regelanwendung auf ein Paar von Literalen bleibt stets eines der beiden Literale übrig[1]. Es genügt daher, alle Paare von Literalen nur genau einmal zu betrachten.

(b) Disjunktionsreduktion

Versuche nun, die Disjunkte durch Anwendung der Disjunktionsreduktions-Regeln für Attributwerte zusammenzufassen (vgl. Unterabschnitt 9.2.2).

Bilde alle Paare von Disjunkten und wende die Regeln an. Bei erfolgreicher Regelanwendung auf ein Paar von Disjunkten bleibt stets eines der beiden Disjunkte übrig[2]. Es genügt daher, alle Paare von Disjunkten nur genau einmal zu betrachten.

Bleibt am Ende der Verarbeitung genau ein $\bot$-Disjunkt übrig, so ist der Attributwert inkonsistent. Da dieser Clash bis auf Graphbeschreibungsebene weitergereicht würde (vgl. Propagierungsregeln des Kalküls), kann die Verarbeitung bereits an dieser Stelle abgebrochen werden.

[1] Man vergleiche dazu alle linken und rechten Seiten der Verarbeitungsregeln. Falls eine Regel zu einem Clash $\bot$ führt, so kann bedingt durch den konjunktiven Kontext das gesamte Disjunkt als Clash deklariert werden.

[2] Man vergleiche dazu alle linken und rechten Seiten der Regeln. Falls eine Regel zu $\top$ führt, so ist aufgrund der Disjunktiven Normalform der gesamte Attributwert mit $\top$ zu belegen. Der Konsistenzcheck ist also erfolgreich beendet.

2. Attributspezifikationen (Verarbeitungsblock: Attributspezifikationen - Konsistenzcheck)

   Untersuche nun in der Attributspezifikations-Tabelle die beiden Einträge für den Typ und die Attributspezifikation:

   (a) Reduktion von Typ-Paaren

   Sollte sich in der Typ-Spalte mehr als eine Typangabe finden, so wende die Typ-Reduktionsregeln auf die Angaben an (vgl. Unterabschnitt 9.3.4). Ergibt eine Regelanwendung einen Clash $\perp$, so deklariere (durch den konjunktiven Kontext) die gesamte Attributspezifikation als Clash. Durch die Clash-Propagierung würde der Clash nun bis zur Graphbeschreibungsebene getragen. Die Verarbeitung kann daher an dieser Stelle abgebrochen werden, das gesamte Anfragedisjunkt wird als Clash deklariert.

   (b) Reduktion von Attribut-Paaren

   Trotz des konjunktiven Kontextes müssen noch Attribut-Wert-Paare zusammengefasst werden, wenn sie sich auf dasselbe Attribut beziehen: `f=#d1 & f=#d2`. Dieser Fall tritt insbesondere beim Verschmelzen eines Korpusknotens mit einer Knotenbeschreibung in der Anfrage auf. In diesem Falle werden die Attributwerte `#d1:d1` und `#d2:d2` durch die Bildung von `d1 & d2` zu einem gemeinsamen Attributwert unifiziert (vgl. Reduktionsregel in Unterabschnitt 9.3.4).

   Genauer: Die Attributwertvariable `#d2` wird in der Umbenennungshalde auf `#d1` abgebildet und die Konjunktion `d1 & d2` der Variablen `#d1` zugeordnet. Anschließend wird für den Attributwert `#d1` erneut ein Konsistenzcheck durchgeführt (siehe Schritt 1). Sollte es dabei zu einem Clash kommen, so ist die Attributspezifikation inkonsistent und wird mit einem Clash markiert. Die Verarbeitung kann abgebrochen werden.

3. Graphbeschreibungsebene (Verarbeitungsregeln: Graphbeschreibungen - Konsistenzcheck)

   Untersuche schließlich alle Knotenrelationen des Disjunkts. Sollte sich eine Relation der Form `#v R #v` unter ihnen befinden, ist das gesamte Disjunkt aufgrund des konjunktiven Kontextes inkonsistent.

Laufzeitanalyse: Der Aufwand auf Attributwert-Ebene beträgt größenordnungsmäßig $n^2+n\cdot m^2$ Paarprüfungen für n Disjunkte und m Konjunkte pro Disjunkt. Auf Attributspezifikationsebene wächst der Aufwand quadratisch mit der Anzahl der Formelbestandteile für Typen und Attribut-Wert-Paare. Ein Hauptteil der Arbeit ist dabei bereits bei der Herbeiführung der Disjunktiven Normalform geleistet worden. Diese wird beim Konsistenzcheck konsequent ausgenutzt.

Beim einmaligen Konsistenzcheck unmittelbar nach der Haldenerzeugung fällt der Aufwand natürlich weniger stark ins Gewicht. Doch sollte berücksichtigt werden, dass der beschriebene Konsistenzcheck einen wesentlichen Bestandteil des Korpusgraph-Tests darstellt (vgl. Unterabschnitt 12.4). Hier besteht also ein evtl. Flaschenhals bei der Verarbeitung. Anfrageoptimierungen, die die Reihenfolge der Disjunkte auf Attributwertebene bzw. Attribut-Wert-Paare auf Attributspezifikationsebene möglichst geschickt anordnen, sind daher für eine effiziente Verarbeitung unbedingt erforderlich (vgl. Kapitel 13.3). Als Faustregel kann festgehalten werden: Je eher eine vorhandene Inkonsistenz aufgedeckt werden kann, desto effizienter die Verarbeitung.

Beispiel

Zur Illustration des Konsistenzchecks wird das Beispiel des vorangegangenen Unterabschnitts wieder aufgegriffen. Durch die Veränderung `case=(#c:"DAT")` für den zweiten Teil der Kasus-Kongruenz wird die Beispielanfrage inkonsistent:

```
#np:[#f1: cat="NP"] &
#t1:[#f2: pos="ADJA" & case=(#c:"NOM"|"AKK")] &
#t2:[#f3: pos="NN" & case=(#c:"DAT")] &
#np > #t1 & #np > #t2 & #t1 . #t2
```

Die Halde zu dieser neuen Anfrage sieht folgendermaßen aus. Man beachte, dass die Konjunktion der beiden Belegungen für `#c` bereits bei der Haldenerzeugung in Disjunktive Normalform umgewandelt wird.

- Knotenbeschreibungen

Knotenvar.	Attr.-Spez.	Knoten-ID
#np	#f1	–
#t1	#f2	–
#t2	#f3	–

- Attributspezifikationen

Attr.-Spez.-Var.	Typ	Attributspezifikation
#f1	NT	cat=#d1
#f2	T	pos=#d2 & case=#c
#f3	T	pos=#d3 & case=#c

- Attributwerte

Attr.-Wert-Var.	Attributwert
#d1	"NP"
#d2	"ADJA"
#d3	"NN"
#c	("NOM"&"DAT") | ("AKK"&"DAT")

- Umbenennungshalde

  Diese Halde ist am Ende der Haldenerzeugung leer.

Beim Konsistenzcheck für die Attributwert-Variable `#c` werden zunächst Inkonsistenzen in beiden Disjunkten festgestellt. Da das Clash-Symbol selbst einen Typ darstellt, greift anschließend die Typ-Disjunktions-Regel `t1 | t2` und es bleibt als Resultat ein Clash übrig. Durch diesen Clash wird die Verarbeitung abgebrochen, die Beispielanfrage ist inkonsistent. Ableitung:

```
("NOM"&"DAT") | ("AKK"&"DAT")
⇒ (⊥) | ("AKK"&"DAT")
⇒ (⊥) | (⊥)
⇒ ⊥
```

12.3.4 Zusammenfassung

Abb. 12.2 fasst zusammen, welcher Punkt des Grobalgorithmus bis zu diesem Zeitpunkt erreicht worden ist. Die ursprüngliche Anfrage q ist durch die Normalisierung in eine Liste von Disjunkten zerfallen. Für jedes Disjunkt ist eine Halde erzeugt und anschließend die Halde zusammen mit dem Disjunkt auf eine mögliche Inkonsistenz untersucht worden. Dabei kann es für einige Disjunkte bereits zu einem Clash gekommen sein. Diese Disjunkte können also niemals aus einem Korpus abgeleitet werden und brauchen deshalb bei der Anfrageverarbeitung nicht mehr berücksichtigt werden. Für die übrigen (konsistenten) Disjunkte erfolgt nun der Test gegen die Korpusgraphen.

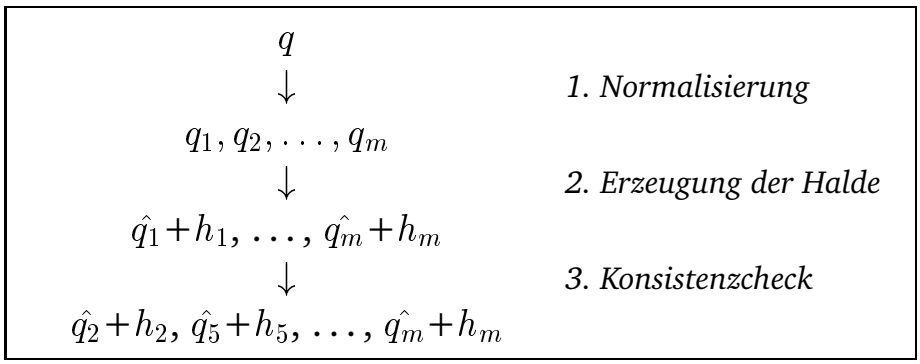


Abbildung 12.2: Zusammenfassung der bisherigen Schritte

12.4 Graphenweise Anfrageevaluation

Zur Beschreibung der graphenweisen Anfrageevaluation wird zunächst das Konzept in Unterabschnitt 12.4.1 in Form eines Grobalgorithmus vorgestellt. Die zentralen Bestandteile des Algorithmus werden anschließend in den Unterabschnitten 12.4.2 bis 12.4.5 genauer ausgeführt: Knotentest, Relationstest, Prädikatstest und die Repräsentation der Anfrageergebnisse. Nach der Illustration des Algorithmus durch ein ausführliches Beispiel in Unterabschnitt 12.4.6 schließt eine Aufwandsabschätzung in Unterabschnitt 12.4.7 den vorliegenden Abschnitt ab.

12.4.1 Grobalgorithmus

Der Ausgangspunkt für die graphenweise Evaluation stellt sich folgendermaßen dar (vgl. Abb. 12.2): Die ursprüngliche Anfrage q ist in Disjunkte q_i zerfallen, die nun unabhängig voneinander verarbeitet werden. Dazu ist für jedes Disjunkt eine Halde h_i erzeugt und ein Konsistenzcheck durchgeführt worden.

Gesucht ist ein Algorithmus, der das Anfragedisjunkt q_i mit Hilfe der Halde h_i gegen einen Korpusgraphen g_j testet. Dieser Algorithmus stellt durch Anwendung von Kalkülverarbeitungsregeln fest, ob es eine Knotenauswahl gibt, bei der für q_i&g_j kein Widerspruch abgeleitet werden kann. Der Korpusgraph g_j, d.h. die in g_j definierten Knotenbeschreibungen, Basisrelationen und abgeleiteten Relationen und Prädikate sind dabei über den Korpusindex zugreifbar.

Steht ein solcher Algorithmus zur Verfügung, kann ein Anfragedisjunkt mit Hilfe dieses Algorithmus nacheinander gegen alle Korpusgraphen getestet werden. Die Einzelergebnisse werden in einer gemeinsamen Ergebnisliste gesammelt.

Idee

Zur Illustration der Algorithmusidee soll das folgende Anfragedisjunkt dienen. Die Haldenerzeugung sei bereits erfolgt.

```
#np & #t1 & #t2 & #np > #t1 & #np > #t2 & arity(#np,2)
```

Die naheliegende Idee besteht darin, dieses Anfragedisjunkt als Konjunktion von Knotenbeschreibungen, Knotenrelationen und Prädikaten von links nach rechts zu durchwandern. Dabei wird versucht, Knotenbeschreibungen zu binden, also mit Graphknoten zu identifizieren. Relationen sowie Prädikate müssen lediglich getestet werden, da die enthalteten Knotenbeschreibungen stets vorher gebunden worden sind. Diese Eigenschaft der Anfrage ist durch die Pre-Normalisierung erreicht worden (vgl. Unterabschnitt 12.2.1).

Ensteht bei einer Knotenbindung, einem Relationstest oder Prädikatstest eine Inkonsistenz, so wird ein Backtracking ausgelöst. Die Verarbeitung wird mit der nächsten Alternative des linken Nachbarn des aktuellen Konjunkts fortgesetzt, d.h. im Falle eines Knotentests wird der nächste Knoten-Kandidat untersucht. Diese Algorithmusidee wird im Folgenden formalisiert.

Algorithmus

Der Algorithmus der graphenweisen Verarbeitung ist auf Seite 221 definiert. Die vier Hauptbausteine des Algorithmus (markiert durch Boxen) sind nur angedeutet und werden in den nachfolgenden Abschnitten detailliert beschrieben.

Der Algorithmus setzt voraus, dass das Anfragedisjunkt aus einer Konjunktion von n Konjunkten (also Knotenbeschreibungen, Knotenrelationen und Prädikaten) besteht. Der Algorithmus besteht aus einer Prozedur *eval*, die die Verarbeitung des Anfragekonjunkts an der Position pos durchführt. Der Aufruf der Prozedur *eval(1)* stellt den eigentlichen Auswertungsalgorithmus dar.

Die Aufgabe der Prozedur *eval(int pos)* besteht darin, die Bindung bzw. den Test des Konjunkts mit der Nummer pos durchzuführen und bei einer positiven Bindung bzw. bei einem positiven Test durch Aufruf von *eval(pos+1)* die Weiterverarbeitung für das nächste Konjunkt zu initiieren. Die Prozedur *eval* ist rekursiv organisiert, der Rücksprung der Verarbeitung an die Stelle des letzten *eval*-Aufrufs (im Falle eines Konflikts oder nach Untersuchung aller Alternativen eines Schrittes) ist als Backtracking realisiert.

Zu Beginn des Algorithmus wird geprüft, ob bereits das Ende des Disjunkts erreicht ist. Ist dies der Fall, so konnten alle Knotenbeschreibungen gebunden werden und alle Tests waren erfolgreich. Damit ist eine Knotenauswahl gefunden, für die q_i&g_j konsistent ist. Nach der Kalküldefinition ist die Anfrage aus dem Korpus bzw. aus dem Korpusgraphen herleitbar.

Die ermittelte Variablenbelegung wird in die Ergebnisliste übertragen. Die Datenstruktur der Ergebnisliste wird in Unterabschnitt 12.4.5 dargestellt. Trotz der erfolgreichen Suche wird die Suche an dieser Stelle nicht abgebrochen, sondern lediglich die Prozedurverarbeitung an dieser Stelle beendet und ein Backtracking ausgelöst. Denn schließlich kann ein Korpusgraph eine Anfrage auch mehrfach erfüllen (z.B. `[cat="NP"]`).

Ist das Ende des Disjunkts noch nicht erreicht, so muss an dieser Stelle das Konjunkt mit der Nummer pos verarbeitet werden. Hier gibt es drei Fälle zu unterscheiden:

(a) Knotenbeschreibung `#v` (Verarbeitungsblock: Knotenbeschreibungen - Knotenauswahl)

Ist die Knotenbeschreibung bereits an einen Knoten gebunden, so gibt es hier nichts zu testen. Die Verarbeitung wird mit dem nächsten Konjunkt durch Aufruf von *eval(pos+1)* fortgesetzt.

Prozedur *eval(int pos)* {

Ist bereits das Ende der Anfrage erreicht, d.h. $pos > n$?

- ja (,d.h. die Suche ist erfolgreich)
  (a) Füge das Suchresultat der Ergebnisliste hinzu.
  (b) return;
- nein
  Verzweige in Abhängigkeit von der Form des Konjunkts:
  (a) Knotenbeschreibung `#v`
  Ist die Knotenbeschreibung `#v` bereits an einen Graphknoten gebunden?
  – ja: Rufe *eval(pos+1)* auf. return;
  – nein:
  i. Bestimme für den Knoten `#v` alle Graphknoten-Kandidaten $v_1, \ldots, v_m$.
  ii. Für alle Kandidaten $v_1, \ldots, v_m$:
  A. push(h);
  B. Führe einen Knotentest für `#v` gegen den Graphknoten v_i auf der Halde h durch.
  C. Ist der Knotentest erfolgreich, so rufe *eval(pos+1)* auf.
  D. pop(h);
  iii. return;
  (b) Knotenrelation `#v R #w`
  i. Führe einen Relationstest für die Relation `#v R #w` durch.
  ii. Ist der Relationstest erfolgreich, so rufe *eval(pos+1)* auf.
  iii. return;
  (c) Prädikat `p(#v)`
  i. Führe einen Prädikatstest für das Prädikat `p(#v)` durch.
  ii. Ist der Prädikatstest erfolgreich, so rufe *eval(pos+1)* auf.
  iii. return;

}

Kehrt die Verarbeitung später durch das Backtracking an diese Stelle zurück, so ist die Prozedur hier beendet, da es keine Alternativen zu betrachten gibt.

Ist die Knotenbeschreibung dagegen noch nicht an einen Knoten gebunden, so besteht die Aufgabe der Prozedur darin, alle Graphknoten zu finden, die mit der Knotenbeschreibung `#v` unifiziert werden können. Mit diesen Knoten ist die Verarbeitung beim nächsten Konjunkt fortzusetzen.

Die nichtdeterministische Knotenauswahl stellt für den Gesamtalgorithmus einen so genannten *choice point* dar. Der nicht-deterministische Schritt wird hier zunächst durch eine erschöpfende Auswahl aller Graphknoten realisiert. Die erschöpfende Vorgehensweise ist dabei natürlich nicht ideal und wird im Rahmen der Effizienzverbesserung in Kapitel 13 weiter verfeinert.

Die Kandidatenmenge für die Korpusgraphen ist damit bestimmt und die Kandidaten können gegen die Knotenbeschreibung `#v` getestet werden. Dieser Knotentest ist in allen Einzelheiten in Unterabschnitt 12.4.2 beschrieben. Ist der Test erfolgreich, so kann das nächste Konjunkt der Anfrage durch Aufruf von *eval(pos+1)* überprüft werden. Kehrt die Anfrageverarbeitung im Rahmen des Backtrackings an diese Stelle zurück oder war der Knotentest erfolglos, so wird der Kandidatentest mit dem nächsten Kandidaten wiederholt.

Wichtig ist, dass sich die Bedingungen in der gegebenen Halde h durch den Knotentest verändern können. Die veränderte Halde h' bildet die Grundlage für alle weiteren Tests in *eval(pos+1)*, d.h. es werden rekursiv alle Möglichkeiten in Bezug auf die durch h' repräsentierte Variablenbelegung untersucht.

Damit das Backtracking die Verarbeitung für den nächsten Kandidaten mit der ursprünglichen Halde fortsetzen kann, muss der aktuelle Status der Halde vor dem Knotentest auf einem Stack zwischengespeichert werden ($\mathsf{push}(h)$). Nach erfolglosem Knotentest bzw. beim Rückkehr der Verarbeitung kann dieser Status dann wieder vom Stack geholt werden ($\mathsf{pop}(h)$).

Dabei macht es aus technischer Sicht wenig Sinn, stets die komplette Halde auf den Stack zu legen, da sich in jedem Verarbeitungsschritt nur ein kleiner Teil der Halde ändern kann. Stattdessen werden in der tatsächlichen Implementation nur die Zustände derjenigen Haldenbestandteile auf den Stack gelegt, die sich auch tatsächlich verändern. Aus diesen Bestandteilen und der veränderten Halde lässt sich dann die ursprüngliche Halde rekonstruieren.

(b) Knotenrelation `#v R #w` (Verarbeitungsblock: Knotenbeschreibungen - Konsistenzcheck - Knotenrelationen)

Durch die Pre-Normalisierung (vgl. Unterabschnitt 12.2.1) ist erreicht worden, dass eine Knotenbeschreibung innerhalb einer Knotenrelation oder innerhalb eines Prädikats nur aus Knotenbeschreibungsvariablen bestehen kann. Die folgende Anfrage beispielsweise ist durch die Pre-Normalisierung entsprechend aufbereitet worden:

```
 #v:[pos="NN" & word="Haus"] > #w  ⇒
#v:[pos="NN" & word="Haus"] & #v > #w
```

Da die Auswertung streng von links nach rechts verläuft, ist eine Knotenbeschreibung innerhalb einer Relation oder eines Prädikats damit zum Zeitpunkt des Relations- bzw. Prädikatstests stets an einen Graphknoten gebunden. Der Test kann damit unmittelbar durchgeführt werden.

Mit Hilfe der Indexdaten wird die Relation zwischen den beiden Knoten `#v` und `#w` geprüft. Dieser Vorgang wird in Unterabschnitt 12.4.3 ausführlich beschrieben.

(c) Prädikat `p(#v)` (Verarbeitungsblock: Knotenbeschreibungen - Konsistenzcheck - Prädikate)

Für ein Prädikat `p` ist die Knotenbeschreibung `#v` ebenfalls bereits gebunden (s.o.). Durch Indexzugriffe kann die Gültigkeit der Prädikatseigenschaft `p(#v)` nachgeprüft werden. Dieser Test ist Unterabschnitt 12.4.4 beschrieben.

12.4.2 Knotentest

Der Knotentest hat die Aufgabe, eine Knotenbeschreibung `#v` mit einem Graphknoten v zu unifizieren. Dazu muss zunächst die Knotenbeschreibung von v aus dem Index erzeugt werden. Der aktuelle Graph sei der k-te Korpusgraph.

- Knotennummer berechnen

  Verwende in Abhängigkeit von der Domäne des Graphknotens v die Indexoperation `int getFirstNTNode(int graphno)` bzw. `int getFirstTNode(int graphno)` zur Berechnung der Startposition des Graphen im Indexkorpus (vgl. Repräsentation der Graphstruktur im Index in Unterabschnitt 11.4.1). Da der Knoten v durch seine Nummer, d.h. die relative Position im Graphen, repräsentiert wird, ist die absolute Nummer des Knotens damit bestimmt.

- Attributwerte bestimmen

  Verwende in Abhängigkeit von der Domäne des Graphknotens v die Indexoperation `String[] get[N]TFeatureNames()` zum Nachschlagen der

Attributnamen für die Domäne des Knotens (vgl. Repräsentation von Korpusinformation in Abschnitt 11.5). Schlage dann mit der Hilfe der Operationen `int get[N]TFeatureValueAt(String feature, int position)` und `String getFeatureValue(String feature, int number)` die Attributwerte für den Knoten v nach (vgl. Index-Attributregister in Unterabschnitt 11.1.1).

Zur Unifikation der beiden Knoten wird die vollständige Knotenbeschreibung von v in die Halde eingewiesen (vgl. Algorithmus zur Haldenerzeugung in Unterabschnitt 12.3.2). Dabei werden die Knotenbeschreibungen von `#v` und v verknüpft, d.h. `#v` wird an die Knoten-ID von v gebunden und die Attributspezifikation von v wird konjunktiv der Attributspezifikation von `#v` hinzugefügt. Anschließend wird ein Konsistenzcheck durchgeführt (vgl. Konsistenzcheck-Algorithmus in Unterabschnitt 12.3.3). Ist dieser Check erfolgreich, so konnten `#v` und v unifiziert werden. Die veränderte Halde repräsentiert das Ergebnis der Verschmelzung.

Das folgende Beispiel illustriert diesen Vorgang. Die Beispielanfrage soll `#np:[cat="NP"]` lauten. Die Halde sieht nach Erzeugung und Konsistenzcheck zunächst wie folgt aus:

- Knotenbeschreibungen

Knotenvar.	Attr.-Spez.	Knoten-ID
`#np`	`#f1`	–

- Attributspezifikationen

Attr.-Spez.-Var.	Typ	Attributspezifikation
`#f1`	`NT`	`cat=#d1`

- Attributwerte

Attr.-Wert-Var.	Attributwert
`#d1`	`"NP"`

Wird nun der Knoten `#v` gegen den Graphknoten `"1":[cat="VP"]` getestet, so entsteht als Ergebnis der Unifikation die folgende Halde:

- Knotenbeschreibungen

Knotenvar.	Attr.-Spez.	Knoten-ID
`#np`	`#f1`	`"1"` (NT)

- Attributspezifikationen

Attr.-Spez.-Var.	Typ	Attributspezifikation
`#f1`	`NT`	`cat=#d1`

- Attributwerte

Attr.-Wert-Var.	Attributwert
#d1	"NP" & "VP"

Der anschließende Konsistenzcheck deckt die Inkonsistenz des Attributwerts `#d1` auf, die Unifikation war erfolglos. Bei einem Test gegen den Graphknoten `"2":[cat="NP"]` hingegen lautet der Attributwert von `#d1` nach der Unifikation `"NP" & "NP"` $\Rightarrow$ `"NP"` und die Verschmelzung ist erfolgreich.

12.4.3 Relationstest

Der Relationstest einer Relation `#v R #w` reduziert sich auf das Nachschlagen der Relation im Index, da die beiden beteiligten Knoten bereits gebunden sind. Die absoluten Knotennummern der beiden Knoten seien bereits ermittelt (vgl. Knoten-Ermittlung beim Knotentest). Abhängig von der Relation `R` können folgende Fälle auftreten:

- Dominanzrelation und Geschwisterrelation (`$`, `>`, `>n` usw.)

  Schlage die Gorn-Adressen von `#v` und `#w` mit Hilfe der Indexoperation `byte[] get[N]TGornAddress(int number)` nach (vgl. Dominanzkodierung im Index in Unterabschnitt 11.3.1). Prüfe die Gültigkeit der Relation anhand der Gorn-Adressen (z.B. mit der Operation `boolean dominatesN(byte[] a, byte[] b, byte n)`).

- Beschriftete Dominanz (`>L`)

  Schlage die Beschriftung der eingehenden Kante von `#w` mit der Operation `String get[N]TFeatureValueAt("edge", int position)` nach (vgl. Kodierung der Kantenbeschriftungen in Unterabschnitt 11.1.3). Stimmt diese Beschriftung mit `L` überein und ist zudem die direkte Dominanzbeziehung zwischen den beiden Knoten erfüllt (s.o.), so ist der Relationstest erfolgreich.

- Präzedenzrelation (`.`, `.n` usw.)

  Schlage die linkesten Nachfolger von `#v` und `#w` mit Hilfe der Indexoperation `int getLeftCorner(int number)` nach (vgl. Kodierung der Präzedenz im Index in Unterabschnitt 11.3.2). Überprüfe anhand der linkesten Nachfolger die Gültigkeit der Relation (z.B. mit der Operation `boolean precedesN(int left1, int left2, byte n)`).

- Geschwister-Präzedenz-Relation (`$.*`)

  Prüfe nach, ob die beiden Knoten in den Relationen `$` und `.*` stehen (s.o.).

- Linke und rechte Ecke (`>@l`, `>@r`)

  Schlage die linke bzw. rechte Ecke von `#v` mit Hilfe der Operation `int get[Left|Right]Corner(int position)` nach bzw. lege für einen Terminalknoten seine Nummer selbst als linke bzw. rechte Ecke fest (vgl. Kodierung der Präzedenz im Index in Unterabschnitt 11.3.2). Stimmt die ermittelte Nummer mit der Nummer von `#w` überein, so ist der Relationstest erfolgreich.

- Sekundäre Kante (`>~`, `>~L`)

  Schlage die Anzahl der sekundären Kanten für den aktuellen Graphen mit der Operation `int getNumberOfSecondaryEdges(int graphno)` nach (vgl. Kodierung sekundärer Kanten im Index in Unterabschnitt 11.4.3). Gehe die sekundären Kanten nun der Reihe nach durch. Stimmen die Nummern der Start- und Zielknoten mit den Nummern von `#v` und `#w` überein und stimmt im Falle der Relation `>~L` zudem die im Index nachgeschlagene Beschriftung mit `L` überein, so ist der Relationstest erfolgreich.

- Negierte Relation (`!R`)

  Prüfe die Gültigkeit der Relation `R`. Die Relation `!R` ist genau dann gültig, wenn `R` nicht gültig ist (s.o.).

Die Vorgehensweise, die Relationsbeziehung zwischen den beiden Knoten `#v` und `#w` zu testen, ersetzt die Regeln des Kalküls, in denen benötigte Relationsaussagen zwischen Graphknoten als transitive Hülle aus den Basisrelationsbeziehungen abgeleitet (vgl. Unterabschnitt 9.5.3) und anschließend per Konsistenzcheck gegen `#v R #w` getestet werden (vgl. Unterabschnitt 9.5.4). Die Ergebnisse dieser Ableitungen sind im Index kompakt ablesbar.

12.4.4 Prädikatstest

Zum Testen eines Prädikats `p(#v)` wird zunächst die Nummer des bereits gebundenen Knotens `#v` nachgeschlagen (vgl. Index-Verwendung beim Knotentest). Für diese Knotennummer wird im Index die zum Prädikatsnamen `p` abgelegte Information abgerufen (vgl. Kodierung von Prädikatseigenschaften in Unterabschnitt 11.4.3). D.h. im Falle von `arity` die Stelligkeit des Knotens mit `int getArity(int position)`, im Falle von `tokenarity` die Tokenstelligkeit des Knotens mit `int getTokenArity(int position)` und im Falle von `(dis)continuous` die (Dis)Kontinuität des Knotens mit `boolean isContinuous(int position)`. Im Falle eines Terminalknotens greifen die Default-Werte `1` für die Stelligkeiten und `true` für die Kontinuität. Es verbleibt der Fall des Prädikats `root`, in dem über die Operation `int getRootNode(int graphno)` die Nummer des Wurzelknotens er-

mittelt wird (vgl. Repräsentation der Graphstruktur im Index in Unterabschnitt 11.4.1).

Im Falle der Kontinuität ist `(dis)continuous(#v)` abgesehen von der Knotenbeschreibung `#v` parameterfrei und es genügt der Test auf Übereinstimmung der booleschen Werte. Im Falle des Wurzelknotens muss geprüft werden, ob die Nummer des Wurzelknotens mit der Knotennummer von `#v` übereinstimmt. In allen übrigen Fällen ist entweder ein Stelligkeitswert oder ein Stelligkeitsintervall angegeben. Hier wird die Übereinstimmung der Werte bzw. die Zugehörigkeit des Indexwerts zum Intervall überprüft.

Die Vorgehensweise, die Knoteneigenschaften von `#v` im Index nachzuschlagen, ersetzt die Regeln des Kalküls, in denen benötigte Prädikatsaussagen über Graphknoten aus dem Korpusgraph abgeleitet (vgl. Unterabschnitt 9.5.3) und anschließend per Konsistenzcheck gegen `p(#v)` getestet werden (vgl. Unterabschnitt 9.5.4).

12.4.5 Anfrageergebnis

In den vorausgegangenen Unterabschnitten ist der Algorithmus zum Test eines Korpusgraphen gegen eine Korpusanfrage detailliert beschrieben worden. Doch welche Informationen repräsentieren überhaupt ein Ergebnis?

Eine große Vereinfachung der Anfrageverarbeitung würde darin bestehen, die Nummern bzw. IDs der matchenden Graphen als Ergebnisrepräsentation festzulegen. Denn in diesem Fall könnte der Auswertungsalgorithmus bereits nach dem Finden des ersten Matches für einen Korpusgraphen stoppen und den nächsten Graphen untersuchen. Doch die Korpusgraph-ID allein reicht für weiterverarbeitende Anwendungen nicht aus.

Eine Alternative besteht darin, die Belegungen sämtlicher Variablen, d.h. die gesamte Halde, zur Repräsentation eines Matches zu verwenden. Doch hier sind letztlich viele Angaben redundant oder nutzlos. Die Attributwerte eines Graphknotens können auch nachträglich im Index nachgeschlagen werden und die Attributspezifikationen sind für spätere Ergebnisverarbeitungen nicht von Interesse. Stattdessen repräsentieren die Paare bestehend aus dem Namen einer Knotenbeschreibungsvariablen und der Knoten-ID, an die sie gebunden ist, einen Match. Mit diesen Information kann der TIGER-XML-Export von Anfrageergebnissen erzeugt werden (vgl. Unterabschnitt 7.2.3).

Die folgende Tabelle stellt einen Beispiel-Match der Anfrage `#np:[cat="NP"] > #t:[pos="NN"]` dar.

Variable	`#np`	`#t`
ID	`"55"` (NT)	`"2"` (T)

Das Gesamtergebnis einer Anfrage besteht dann aus einer Liste solcher Ergebnistabellen. Man beachte, dass ein Graphknoten in einer solchen Liste nur

dann eindeutig bezeichnet werden kann, wenn seine absolute Knotennummer oder seine Graphnummer und (relative) Knotennummer angegeben ist. Aus Vereinfachungsgründen besteht damit die Repräsentation eines Anfrageergebnisses aus einer Liste von matchenden Graphnummern, zu denen jeweils eine Liste von Matchergebnissen angegeben ist. Zur obigen Anfrage könnte das vollständige Ergebnis also beispielsweise folgendermaßen aussehen:

- Graph Nr. 1 (ID s4)

Variable	#np	#t
ID	"55" (NT)	"2" (T)

- Graph Nr. 13 (ID s17)

Variable	#np	#t
ID	"55" (NT)	"2" (T)

Es sei noch bemerkt, dass es Variablen geben kann , die nur in einem Teil der Disjunkte gebunden werden können. Dies ist beispielsweise dann der Fall, wenn eine Anfrage im Laufe der Normalisierung in mindestens zwei Disjunkte zerfällt. In diesem Falle werden vor der Anfrageevaluation die Namen aller Knotenvariablen aufgesammelt, von denen dann im Kontext eines Matches nicht mehr alle gebunden sein müssen. Bei der Anfrage `#np:[cat="NP"] | #t:[pos="NN"]` beispielsweise könnte das Anfrageergebnis folgendermaßen aussehen:

- Graph Nr. 2 (ID s5)

Variable	#np	#t
ID	"45" (NT)	–

- Graph Nr. 7 (ID s11)

Variable	#np	#t
ID	–	"3" (T)

12.4.6 Eine Beispielverarbeitung

Die folgende Beispielverarbeitung illustriert die Vorgehensweise des Algorithmus. Als Anfragebeispiel dient die folgende Graphbeschreibung:

```
#phrase:[cat="NP"] & #token:[pos="ART" & word="ein"] &
#phrase > #token
```

Die Normalisierung der Anfrage, Erzeugung der Halde und Durchführung eines Konsistenzchecks führen zu folgender Halde:

- Knotenbeschreibungen

Knotenvar.	Attr.-Spez.	Knoten-ID
#phrase	#f1	–
#token	#f2	–

- Attributspezifikationen

Attr.-Spez.-Var.	Typ	Attributspezifikation
#f1	NT	cat=#d1
#f2	T	pos=#d2 & word=#d3

- Attributwerte

Attr.-Wert-Var.	Attributwert
#d1	"NP"
#d2	"ART"
#d3	"ein"

Das verbleibende Anfrageskelett sieht dann folgendermaßen aus:

```
#phrase & #token & #phrase > #token
```

Der Evaluationsalgorithmus, der die verbleibende Anfrage mit der Halde gegen einen Korpusgraphen testet, wird nun an einem Ein-Satz-Korpus illustriert, das aus dem in Abb. 12.3 abgebildeten Graphen besteht. Eine vollständige Visualisierung des Index würde an dieser Stelle zu weit führen. Stattdessen stelle man sich den Graphen als Index vorbereitet und die Graphknoten von links nach rechts bzw. von unten nach oben durchnummeriert vor (1T - ART, 2T - NN 3T - VVFIN, 1NT - NP, 2NT - S).

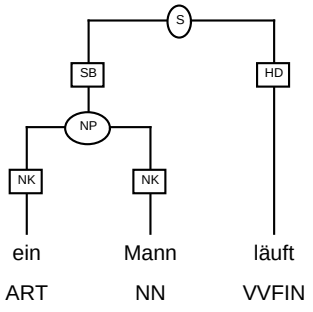


Abbildung 12.3: Der Beispielgraph

Das Anfrageskelett wird nun schrittweise mit dem Backtracking-Algorithmus verarbeitet. Die Schrittnummer i entspricht dabei dem Aufruf der Prozedur *eval(*i*)*.

1. Schritt: `#phrase` binden - a) `NP`-Knoten

Der erste Schritt hat die Aufgabe, die Knotenbeschreibung `#phrase` an einen Graphknoten zu binden. Zur Vereinfachung der Darstellung werden hier nur die beiden inneren Knoten getestet. Es sei dabei angenommen, dass der `NP`-Knoten zuerst getestet wird.

Die Knotenbeschreibung des `NP`-Graphknotens `"1":[cat="NP"]` wird aus dem Index ausgelesen und nun in die Halde eingewiesen. Dabei werden der Graphknoten und `#phrase` unifiziert.

Es sei angemerkt, dass hier mehrere Schritte zu einem zusammengefasst werden. Strenggenommen müsste zunächst der Graphknoten unter einer neuen Knotenbeschreibungsvariable in die Halde eingewiesen werden, um diese neue Knotenbeschreibung anschließend mit `#phrase` zu verschmelzen. Da die Knotenbeschreibung des Graphknotens aber im weiteren Verlauf nicht mehr benötigt wird, wird dieser Zwischenschritt und damit auch der Speicherplatz für einen zusätzlichen Haldeneintrag eingespart.

- Knotenbeschreibungen

Knotenvar.	Attr.-Spez.	Knoten-ID
`#phrase`	`#f1`	`"1"` (NT)
`#token`	`#f2`	–

- Attributspezifikationen

Attr.-Spez.-Var.	Typ	Attributspezifikation
`#f1`	`NT`	`cat=#d1`
`#f2`	`T`	`pos=#d2 & word=#d3`

- Attributwerte

Attr.-Wert-Var.	Attributwert
`#d1`	`"NP" & "NP"`
`#d2`	`"ART"`
`#d3`	`"ein"`

Der anschließende Konsistenztest fasst den Attributwert `"NP" & "NP"` von `#d1` zu `"NP"` zusammen. Der Knotentest ist erfolgreich, die Verarbeitung kann mit dem nächsten Konjunkt fortgesetzt werden.

2. Schritt: `#token` binden - a) Token *ein*

Im zweiten Schritt wird versucht, den Knoten `#token` zu binden. Zur Vereinfachung der Darstellung werden hier nur die drei Terminalknoten des Graphen getestet. Die drei Knoten werden dabei in ihrer Tokenreihenfolge verarbeitet.

Die Unifikation von `#token` und dem ersten Graphknoten (Token *ein*) führt zu folgender Haldenbelegung:

- Knotenbeschreibungen

Knotenvar.	Attr.-Spez.	Knoten-ID
`#phrase`	`#f1`	`"1"` (NT)
`#token`	`#f2`	`"1"` (T)

- Attributspezifikationen

Attr.-Spez.-Var.	Typ	Attributspezifikation
`#f1`	`NT`	`cat=#d1`
`#f2`	`T`	`pos=#d2 & word=#d3`

- Attributwerte

Attr.-Wert-Var.	Attributwert
`#d1`	`"NP"`
`#d2`	`"ART" & "ART"`
`#d3`	`"ein" & "ein"`

Der anschließende Konsistenzcheck vereinfacht die Attributwerte zu jeweils einer Konstante. Der Knotentest ist erfolgreich, die Verarbeitung kann mit dem nächsten Konjunkt fortgesetzt werden.

3. Schritt: Relation `#phrase > #token` testen

Im Index wird nun nachgeschlagen, ob der nicht-terminale Knoten `"1"` den terminalen Knoten `"1"` direkt dominiert. Dies ist der Fall, der Relationstest ist erfolgreich. Die Verarbeitung kann mit dem nächsten Konjunkt fortgesetzt werden.

4. Schritt: Match gefunden

Da alle Konjunkte bearbeitet wurden, ist ein Match gefunden worden. Die Belegungen der Knotenbeschreibungsvariablen werden der Ergebnisliste hinzugefügt:

- Satz Nr. 1

Variable	`#phrase`	`#token`
ID	`"1"` (NT)	`"1"` (T)

Das Backtracking führt zur Weiterführung der Verarbeitung bei Schritt 3. Da dort lediglich ein Relationstest durchgeführt wird, ist die Verarbeitung des Schrittes beendet. Das Backtracking führt zur Fortsetzung der Verarbeitung bei Schritt 2.

2. Schritt: `#token` binden - b) Token *Mann*

Nachdem das Token *ein* bereits getestet wurde, geht es nun mit dem nächsten Kandidaten weiter. Dazu wird zunächst die Halde vom Stack geholt. Sie hat wieder den Stand wie beim ersten Übergang zu Schritt 2. Die Reinitialisierung der Halde stellt den Abschluss des Knotentests für das Token *ein* dar.

Die Knotenbeschreibung des Graphknotens wird aus dem Index ausgelesen, in die Halde eingefügt und dabei mit `#token` unifiziert. Es entsteht die folgende Halde:

- Knotenbeschreibungen

Knotenvar.	Attr.-Spez.	Knoten-ID
`#phrase`	`#f1`	`"1"` (NT)
`#token`	`#f2`	`"2"` (T)

- Attributspezifikationen

Attr.-Spez.-Var.	Typ	Attributspezifikation
`#f1`	`NT`	`cat=#d1`
`#f2`	`T`	`pos=#d2 & word=#d3`

- Attributwerte

Attr.-Wert-Var.	Attributwert
`#d1`	`"NP"`
`#d2`	`"ART" & "NN"`
`#d3`	`"ein" & "Mann"`

Der anschließende Konsistenzcheck deckt einen Clash im Attributwert `#d2` bzw. `#d3` auf, der Knotentest ist erfolglos. Die Verarbeitung wird beim nächsten Kandidaten fortgesetzt. Zuvor wird die Halde vom Stack geholt.

2. Schritt: `#token` binden - c) Token *läuft*

Der Knotentest für das Token *läuft* ist ebenfalls erfolglos. Das Backtracking führt zur Fortsetzung des Verfahrens bei Schritt 1.

1. Schritt: `#phrase` binden - b) `S`-Knoten

Die Verarbeitung im 1. Schritt setzt sich mit dem `S`-Knoten fort. Der Knotentest ist für diesen Knoten erfolglos. Das Backtracking beendet die Algorithmus-Verarbeitung.

Ergebnis

Als Ergebnisliste ist ermittelt worden:

- Satz Nr. 1

Variable	#phrase	#token
ID	"1" (NT)	"1" (T)

12.4.7 Aufwandsabschätzung

Die normalisierte Anfrage enthalte n namentlich verschiedene Knotenbeschreibungen (bei Knoten mit demselben Namen erfolgt durch vorherige Unifikation nur ein Bindungsvorgang) und m Relationen bzw. Prädikate. Das Korpus umfasse N Korpusgraphen und der größte Korpusgraph enthalte K Knoten.

Da es für jede Knotenbeschreibung K potentielle Knotenkandidaten gibt, entstehen K^n mögliche Belegungen der n Knoten. Für jede der K^n möglichen Belegungen sind bis zu n Knotentests und bis zu m Prädikats- oder Relationstests durchzuführen. Damit setzt sich der Aufwand des Evaluationsalgorithmus zum Prüfen aller Korpusgraphen aus bis zu $N \cdot K^n \cdot n$ Knotentests und $N \cdot K^n \cdot m$ Relations- und Prädikatentests zusammen.

Der Beitrag der Knotentests ist dabei dominant, da ein Knotentest mit einer Haldenzwischenspeicherung, einer Haldeneinweisung und einem Konsistenzcheck deutlich aufwändiger ist als ein Relations- bzw. Prädikatstests.

Der prinzipiell exponentielle Aufwand ist als hoch einzustufen. Allerdings ist in diesem Kapitel aus Gründen der Übersicht lediglich der Basisalgorithmus beschrieben worden. Verfeinerungen des Algorithmus im Rahmen des nachfolgenden Kapitels werden die Laufzeit für viele Fälle drastisch reduzieren.

12.5 Korrektheit und Vollständigkeit

In Kapitel 11 ist die Korpusrepräsentation im Index beschrieben worden, der die Ergebnisse eines großen Teils der Kalkül-Verarbeitungsregeln repräsentiert. Das vorliegende Kapitel hat die Anfrageverarbeitung zur Laufzeit behandelt. Damit ist die Implementation des Kalküls vollständig beschrieben.

Es bleibt die Frage, ob die Verarbeitungsweise des Kalküls korrekt und vollständig realisiert worden ist. Dazu ist nachzuweisen, dass eine Anfrage q aus einem Korpus C genau dann in der Implementation `I` herleitbar ist, wenn die Anfrage q aus dem Korpus C auch im Kalkül `K` herleitbar ist, d.h.

$C \vdash_{\mathtt{K}} q \quad \Leftrightarrow \quad C \vdash_{\mathtt{I}} q$.

Dieser Nachweis wird an dieser Stelle nicht formal, sondern in Form einer Beweisskizze geführt. Der Nachweis wird in drei Schritten erbracht.

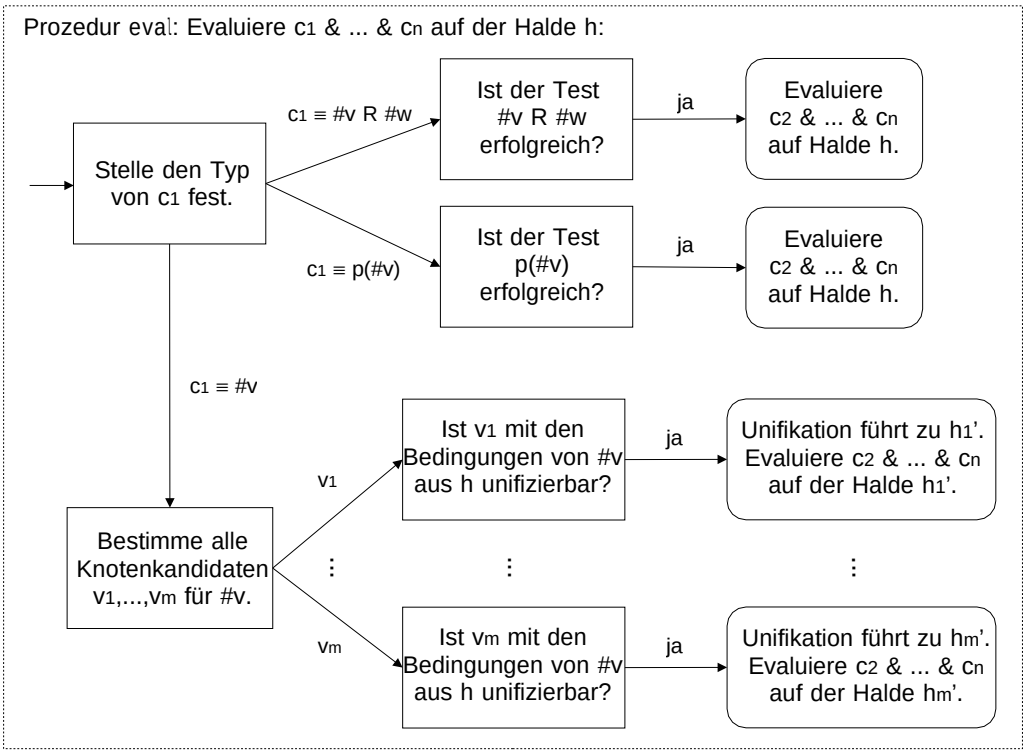


Abbildung 12.4: Illustration der Prozedur *eval*

Um die Arbeitsweise des Algorithmus für den Nachweis noch einmal zu verdeutlichen, illustriert Abb. 12.4 die Verarbeitung der zentralen Prozedur *eval* für einen Verarbeitungsschritt. Aus dieser Darstellung wird unmittelbar deutlich, dass die noch zu untersuchende Formel im Falle eines positiven Knotentests, Relationstests oder Prädikatstests auf der nächsten Rekursionsebene stets um ein Konjunkt kürzer wird. Im negativen Falle fällt der aktuell verarbeitete Verarbeitungszweig komplett weg und durch das Backtracking wird die Verarbeitung auf der aufrufenden Ebene fortgesetzt. Damit ist unmittelbar einsichtig, dass der Algorithmus terminiert.

1. Schritt: Voraussetzungen

Es wird für die beiden Schritte 2 und 3 vorausgesetzt, dass (a) alle Verarbeitungsregeln vollständig implementiert worden sind und (b) die Implementation für jede Regel in korrekter Weise erfolgt ist.

a) Umsetzung aller Regeln

Durch die expliziten Verweise der Implementationsblöcke auf die entsprechenden Verarbeitungsblöcke im Kalkül im Index und in der Laufzeit-Verarbeitung kann nachgewiesen werden, dass alle Regeln des Kalküls berücksichtigt werden.

b) Bedeutungsäquivalenz der Datenstrukturen

Der Index repräsentiert die Korpusbeschreibungen aller Korpusknoten und aller (abgeleiteten) Relationen und Prädikate über den Korpusknoten. Die Korrektheit der Ableitungen ergibt sich aus den Darstellungen der Algorithmen zur Konstruktion der Index-Bestandteile.

Die anfrageabhängigen Verarbeitungsregeln des Kalküls sind direkt blockweise umgesetzt worden (vgl. Verweise der einzelnen Blöcke auf die Kalkülblöcke). Variablenbindungen werden durch eine Implementation des Haldenmodells repräsentiert, die das Modell direkt implementiert. Zwar sind die Datenstrukturen von Korpusanfrage (Anfrageskelett und Halde) und Korpusdefinition (Index) verschieden, doch bildet die Einweisung eines Korpusknotens in die Halde eine saubere Schnittstelle.

2. Schritt: Widerlegungsvollständigkeit der Constraint-Reduktion

In diesem Schritt wird gezeigt, dass eine Ableitung in der Implementation $\mathtt{I}$ genau dann zu einem Widerspruch führt, wenn auch im Kalkül $\mathtt{K}$ ein Widerspruch herleitbar ist, d.h.

$g \vdash_{\mathtt{K}} \bot \Leftrightarrow g \vdash_{\mathtt{I}} \bot$.

Eine zu testende Graphbeschreibung g steht hier stellvertretend für die zu testende Konjunktion einer Anfrage q mit einem Korpusgraphen g', d.h. g entspricht $q \& g'$.

a) $g \vdash_{\mathtt{K}} \bot \Rightarrow g \vdash_{\mathtt{I}} \bot$

Angenommen, es gibt eine Graphbeschreibung g mit $g \vdash_{\mathtt{K}} \bot$, aber **nicht** $g \vdash_{\mathtt{I}} \bot$. Mögliche Ursachen für diesen Konflikt sind:

- Eine Regel, die die Inkonsistenz aufgedeckt hätte, ist nicht oder nicht korrekt implementiert worden.

  Dies ist nach Schritt 1 nicht möglich.

- Die Anwendung einer Regel, die die Inkonsistenz aufgedeckt hätte, ist durch den Verarbeitungsalgorithmus ausgeschlossen worden.

Die Beschreibung des Konsistenzcheck-Algorithmus zeigt die Parallelität der Implementation zum Konsistenzcheck des Kalküls. Es werden keine Verarbeitungsschritte ausgeschlossen, die eine Inkonsistenz aufdecken könnten. Die Wahl der Abfolge der Regelanwendungen innerhalb der Implementation führt zum selben Gesamtergebnis wie jede andere mögliche Verarbeitungsabfolge.

b) $g \vdash_{\mathtt{I}} \bot \Rightarrow g \vdash_{\mathtt{K}} \bot$

Angenommen, es gibt eine Graphbeschreibung g mit $g \vdash_{\mathtt{I}} \bot$, aber **nicht** $g \vdash_{\mathtt{K}} \bot$. D.h. der Algorithmus weist eine Inkonsistenz nach, die nicht gegeben ist. Für einen solchen Konflikt gibt es die folgenden Ursachen:

- Eine Verarbeitungsregel ist nicht korrekt implementiert und führt zur Ableitung der Inkonsistenz.

  Dies ist nach Schritt 1 nicht möglich.

- Durch eine unvollständige Knotenauswahl ist ein Graphknoten übersehen worden, der zu einer konsistenten Ableitung geführt hätte.

  Durch eine erschöpfende Suche bei der Umsetzung dieses nichtdeterministischen choice points wird ein Ausschluss eines Korpusknotens verhindert. Alle Korpusknoten eines Graphen werden berücksichtigt (vgl. Abb. 12.4).

3. Schritt: Korrektheit und Vollständigkeit der Implementation

Es bleibt nachzuweisen, dass eine Anfrage q aus einem Korpus C genau dann in der Implementation $\mathtt{I}$ herleitbar ist, wenn die Anfrage q aus dem Korpus C auch im Kalkül $\mathtt{K}$ herleitbar ist, d.h.

$C \vdash_{\mathtt{K}} q \Leftrightarrow C \vdash_{\mathtt{I}} q$.

a) $C \vdash_{\mathtt{K}} q \Rightarrow C \vdash_{\mathtt{I}} q$

Angenommen, es gibt ein Korpus C und eine Anfrage q mit $C \vdash_{\mathtt{K}} q$, aber **nicht** $C \vdash_{\mathtt{I}} q$. Mögliche Ursachen dafür sind:

- Eine Ableitungsregel fehlt oder ist nicht korrekt implementiert worden.

  Dies ist nach Schritt 1 nicht möglich.

- Durch eine unvollständige Suche ist ein Korpusgraph übersehen worden, der zur Ableitung von q geführt hätte.

  Durch eine erschöpfende Suche bei der Umsetzung dieses nichtdeterministischen choice points wird ein Ausschluss eines Korpusgraphen verhindert. Alle Graphen eines Korpus werden berücksichtigt (vgl. Abb. 12.4).

b) $C \vdash_{\mathtt{I}} q \Rightarrow C \vdash_{\mathtt{K}} q$

Angenommen, es gibt ein Korpus C und eine Anfrage q mit $C \vdash_{\mathtt{I}} q$, aber **nicht** $C \vdash_{\mathtt{K}} q$.

Die Implementation weist also eine Ableitung nach, die eigentlich widerspruchsvoll ist. D.h. die Implementation erkennt eine Inkonsistenz nicht.

Dies ist nach Schritt 2 ausgeschlossen.

Zusammenfassung

Die korrekte und vollständige Arbeitsweise der Implementation ist damit begründet. Ein fehlerfrei arbeitendes Computerprogramm ist allerdings ein unrealistischer Anspruch. Jedoch trägt die zweigeteilte Architektur mit Kalkül und Implementation dazu bei, konzeptuelle und technische Programmierfehler schneller zu beheben.

Kapitel 13

Effiziente Verarbeitung

Im vorangegangenen Kapitel ist der Basisalgorithmus zur Verarbeitung von Korpusanfragen beschrieben worden. Dieser Algorithmus erfüllt zwar seine Aufgabe, doch ist das Laufzeitverhalten noch verbesserungsfähig. Das vorliegende Kapitel entwickelt Ansätze zur Steigerung bzw. Sicherstellung der Effizienz.

In Abschnitt 13.1 wird der Verarbeitungsalgorithmus zur Laufzeit noch einmal analysiert. Dabei werden Ansatzmöglichkeiten für eine Effizienzverbesserung herausgearbeitet. Als erste Realisierung dieser Ansatzpunkte werden in Abschnitt 13.2 Suchraumfilter entwickelt, die auf fortgeschrittenen Indexierungstechniken beruhen und die Menge der zu testenden Korpusgraphen reduzieren. Im Rahmen der Anfrageoptimierung werden in Abschnitt 13.3 vor allem Umordnungen der Anfrage vorgenommen, die sich erheblich auf die Laufzeit auswirken.

Als Ergänzung werden in Abschnitt 13.4 auch rein technische Ansätze verfolgt, die die Möglichkeiten der verwendeten Programmiersprache Java für eine effizientere Verarbeitung ausnutzen. Der abschließende Abschnitt 13.5 zeigt auf, wie die insgesamt erzielte Effizienz der Implementation im Vergleich zu anderen Baumbank-Suchwerkzeugen zu beurteilen ist und fasst diesen Teil der Arbeit mit der resultierenden Gesamtarchitektur der Anfrageverarbeitung zusammen.

13.1 Ansätze

Die Aufwandsabschätzung der Anfrageverarbeitung zur Laufzeit in Unterabschnitt 12.4.7 hat gezeigt, dass der Aufwand linear mit der Anzahl der Korpusgraphen N und exponentiell mit der Anzahl n der Knotenbeschreibungen in der Anfrage wächst. Dabei bezeichnet K die Anzahl der Knoten für den größten Korpusgraphen, woraus sich die Anzahl der möglichen Knotenbelegungen K^n ergibt. Bei dieser Abschätzung wird jedoch der denkbar schlechteste Fall angenommen.

Der nachfolgende Unterabschnitt 13.1.1 zeigt auf, dass dieser Fall in der Praxis nur selten auftritt. Er skizziert zudem Ansatzmöglichkeiten, die zu einer weiteren Reduzierung der Laufzeit führen können. Diese Ansätze werden in den nachfolgenden Abschnitten in die Tat umgesetzt. Unterabschnitt 13.1.2 definiert anschließend den um die Optimierungsansätze erweiterten Verarbeitungsalgorithmus.

13.1.1 Analyse des Algorithmus

Der Aufwand für den Auswertungsalgorithmus ist als hoch einzustufen. Die folgenden Beobachtungen relativieren allerdings dieses Urteil und zeigen zugleich Ansatzmöglichkeiten für Verbesserungen auf.

- Erfolglose Suche

  Für einen Korpusgraphen werden nur dann n Knotenbeschreibungen gebunden, wenn die Suche erfolgreich ist, d.h. eine konsistente Variablenbelegung gefunden wird. Für die bis zu K^n denkbaren Variablenbelegungen ist dies aber nur für einen Bruchteil der Fall. Man führe sich zum Beispiel vor Augen, dass selbst für eine sehr generelle Anfrage wie `[cat="NP"] > [cat="VP"]` in einem typischen Korpusgraph nur wenige NP- und VP-Knoten zu finden sind. Die Verarbeitung führt daher nur in wenigen Fällen überhaupt zum zweiten Knoten, weshalb die exponentielle Laufzeit nur im schlechtesten Fall auftreten kann. In der Praxis scheitern die meisten Verarbeitungspfade bereits an den ersten beiden Konjunkten.

- Anfrageoptimierung

  Ist eine Anfrage sehr speziell, so wird sie nur von wenigen Graphen erfüllt werden. Sind die Formelbestandteile des Anfrageskeletts und der Haldenbedingungen nun in einer geschickten Reihenfolge angeordnet (vgl. Anfrageoptimierungsstrategien in Abschnitt 13.3), so wird die Anfrageauswertung bereits an einem frühen Konjunkt scheitern und der volle rekursive Abstieg eher die Ausnahme sein.

- Flaschenhals *Knotenauswahl*

  Die Abschätzung von K^n Knotentests trifft nur dann zu, wenn stets alle K Knoten gegen die n Knotenbeschreibungen getestet werden.

  In den meisten Fällen ist allerdings die Domäne einer Knotenbeschreibung `#v` bereits bekannt (`T`, `NT` oder `FREC`). Diese Domäne kann in der Attributspezifikationstabelle in der Halde nachgeschaut werden, da eine Knotenbeschreibung `#v` stets an eine Attributspezifikations-Variable gebunden ist. Ist die Domäne `T` bzw. `NT`, so kommen lediglich alle Terminal- bzw. Nichtterminalknoten als Kandidaten in Frage. Denn ansonsten käme

es ohnehin beim Test des Knotens (vgl. Unterabschnitt 12.4.2) zu einem Typkonflikt. Damit reduziert sich die Kandidatenmenge bereits erheblich.

Eine sehr effektive Heuristik zur weiteren Reduzierung der Knotentest-Kandidatenmenge wird im Rahmen der Anfrageoptimierung verfolgt (vgl. Abschnitt 13.3).

- Filterstrategien

  Sehr vielversprechend sind Verfahren zur Filterung des Suchraums. Sie reduzieren anhand der Anfrage und der Halde die Korpusgraph-Kandidaten, die für eine Anfrage überhaupt in Frage kommen. Damit muss nur noch ein Teil der Korpusgraphen getestet werden, d.h. die Zahl der Korpusgraphen N kann reduziert werden. Filterstrategien werden in Abschnitt 13.2 entwickelt.

13.1.2 Erweiterung des Algorithmus

1. Parse die Anfrage q. $(\star)$
2. Pre-Normalisiere die Anfrage q. $(\star)$
3. Überführe die Anfrage q in Disjunktive Normalform. $(\star)$
4. Für jedes Disjunkt $q_1, \ldots, q_m$:
   (a) Erzeuge die Halde h zum Disjunkt q_i. $(\star)$
   (b) Wende den Suchraumfilter auf das Disjunkt q_i an. $\bigtriangledown$
   (c) Optimiere die Halde h. $\triangleright$
   (d) Optimiere das Disjunkt q_i. $\triangleright$
   (e) Für jeden Korpusgraphen $g_1, \ldots, g_n$ im Suchraum:
   - Teste das Disjunkt q_i mit der Halde h auf dem Korpusgraphen g_j. $(\star)$
5. Füge die Anfrageergebnisse der Disjunkte zusammen. $(\star)$

Abbildung 13.1: Verarbeitung von Korpusanfragen - Grobalgorithmus
Legende: $(\star)$ - Kernalgorithmus; $\bigtriangledown$ - Suchfilter; $\triangleright$ - Anfrageoptimierung

Abb. 13.1 zeigt den erweiterten Grobalgorithmus (vgl. auch Abschnitt 12.1), der die Anfrageverarbeitung um die skizzierten Bausteine Anfrageoptimierung und Suchraumfilter zur Effizienzsteigerung ergänzt. In den folgenden Abschnitten werden diese Bausteine detailliert beschrieben.

13.2 Suchraumfilter

In diesem Abschnitt geht es um die Implementation so genannter Suchraumfilter. Unter einem Suchraumfilter wird ein Verfahren verstanden, das den Suchraum für die Anfrageverarbeitung (d.h. die Menge der zu testenden Korpusgraphen) einschränken soll. Das Verfahren basiert meist auf Index-Daten. Ein solcher Filter stellt damit eine geschickte Realisierung der nicht-deterministischen Korpusgraph-Auswahl des Kalküls dar. Bei der Umsetzung eines Filters ist jedoch sicherzustellen, dass nur solche Korpusgraphen aussortiert werden, die tatsächlich nicht für einen Match in Frage kommen. Ansonsten ist die Vollständigkeit der Gesamtimplementation nicht mehr gegeben.

Kann ein Suchraumfilter in sehr kurzer Zeit angewendet werden, fallen seine zusätzlichen Kosten nicht ins Gewicht. Stattdessen werden seine Kosten durch die eingesparte Suche, die sonst auf dem ausgefilterten Suchraum angefallen wäre, wieder wettgemacht. Die im Rahmen eines Suchraumfilters verwendeten Index-Ansätze ergänzen die Repräsentation des Korpus als Index.

Ein naheliegender erster Zugang besteht darin, die Basiseinheiten des Datenmodells isoliert zu betrachten und die daraus resultierenden Daten in einem invertierten Korpus abzulegen. Damit spielt die eigentliche Komplexität des Datenmodells zunächst keine Rolle. Zudem sind Standardmethoden des Information Retrieval anwendbar, die ursprünglich zur Textsuche entwickelt worden sind (vgl. Baeza-Yates und Ribeiro-Neto 1999, Kapitel 8). Eine solche Vorgehensweise ist auch in anderen Arbeiten anzutreffen, die auf komplexen Datenmodellen arbeiten (vgl. Arbeiten zu semistrukturierten Daten, z.B. McHugh et al. 1998). Die entstehenden Basis-Indexe stellen hier die Grundlage zum Aufbau komplexerer Index-Ansätze dar.

Liegt der Fokus auf der Indexierung des vorliegenden Datenmodells, so sind zunächst die Arbeiten zur Suche auf semistrukturierten Daten und XML-Daten relevant (vgl. Abiteboul et al. 2000, Abschnitt 8.2.2 und McHugh et al. 1998). Auch die Teilbereiche des Information Retrieval, die sich mit vorstrukturierten Daten wie beispielsweise WWW-Seiten befassen, machen massiven Gebrauch von Baum- bzw. Graph-Indexierungstechniken und sind deshalb von Interesse (vgl. Meuss und Strohmaier 1999a, b). Ein bereits speziell für Baumbanken entwickeltes Konzept findet sich in Argenton (1998). Hier wird die Idee des N-Gramm-Filters zur Textindexierung als sogenannter Baum-Gramm-Filter auf Baumstrukturen übertragen und anschließend für gerichtete azyklische Graphen verallgemeinert.

Der folgende Unterabschnitt 13.2.1 beschreibt eine erste Filterstrategie, die auf der Ebene der Knoten operiert. Der nachfolgende Unterabschnitt 13.2.2 zeigt auf, wie diese elementare Indexierung auch auf Knotenrelationen ausgedehnt werden kann. Weiterführende Arbeiten zur Graphenindexierung, die bestehende Ansätze wie etwa den Baum-Gramm-Filter integrieren, stehen noch aus.

13.2.1 Ein Knotenfilter

Idee

Für die Anwendung des Knotenfilters sei die Anfrage bereits in Disjunktiver Normalform. Dem Knotenfilter liegt die Feststellung zugrunde, dass ein Korpusgraph zur Erfüllung eines Anfragedisjunkts alle Konjunkte erfüllen muss. Unter den Konjunkten nehmen die Knotenbeschreibungen eine Sonderstellung ein, da sie die meisten Restriktionen ausdrücken. In der folgenden Anfrage ist beispielsweise die rechte Knotenbeschreibung sehr restriktiv, da die spezifizierte Wortform *Haus* selten ist:

```
[cat="NP"] > [word="Haus" & pos="NN"]
```

Die Idee des Knotenfilters besteht darin, die Anfrage durch gezieltes Streichen von Information so zu vereinfachen, dass am Ende eine allgemeinere, aber dennoch restriktive Anfrage verbleibt. Dazu werden in einem ersten Schritt alle Relationssymbole durch Konjunktionen ersetzt:

```
[cat="NP"] & [word="Haus" & pos="NN"]
```

Man beachte, dass durch diese Umformung die Semantik des Ausdrucks verallgemeinert wird. Allerdings muss jeder Korpusgraph, der die erste Anfrage erfüllt, auch die zweite Anfrage erfüllen. Die Erfüllbarkeit der vereinfachten Anfrage wird also zur notwendigen Bedingung für die Erfüllbarkeit der ursprünglichen Anfrage. Durch dieses Prinzip wird die Vollständigkeit des Verfahrens sichergestellt.

Diese Strategie wird nun weiter fortgesetzt und die Knoten werden einzeln untersucht. Der *NP*-Knoten ist dabei wenig restriktiv, da `cat="NP"` für sehr viele Korpusknoten gilt. Es bleibt also nur noch ein Knoten übrig.

```
[word="Haus" & pos="NN"]
```

Der verbleibende Knoten enthält eine sehr restriktive Bedingung `word="Haus"` und eine wenig restriktive Bedingung `pos="NN"`. Da die Attributspezifikation in einem konjunktiven Kontext steht, kann der Knoten weiter vereinfacht werden.

```
[word="Haus"]
```

Damit ist die ursprüngliche Anfrage zu einer Ein-Knoten-Anfrage verallgemeinert worden. Können nun die Nummern derjenigen Korpusgraphen nachgeschaut werden, in denen mindestens ein Graphknoten mit der Wortform *Haus* vorkommt, so lässt sich der Korpus-Suchraum reduzieren. Für das Negra-Korpus reduziert sich der Suchraum in diesem Beispiel von 20.600 Korpusgraphen auf 88 Graphen. Die Verarbeitungseffizienz vervielfacht sich also etwa um den Faktor 250, wenn die Kosten der Filterung unberücksichtigt bleiben.

Die beiden folgenden Unterabschnitte spezifizieren den Knotenfilter-Algorithmus. Der nachfolgende Unterabschnitt zeigt die Erweiterung des Index um das sogenannte invertierte Korpus, das das Nachschlagen aller Korpusgraphen zu einem gegebenen Attribut-Wert-Paar erlaubt. Der anschließende Unterabschnitt formalisiert den Algorithmus und stellt eine erste Kosten-Nutzen-Rechnung auf.

Indexerweiterung - Invertiertes Korpus

Ein invertiertes Korpus dreht die gewohnte Sichtweise auf ein Korpus bewusst um: Statt einem Graphen bzw. einem Knoten seine Attributwerte zuzuordnen, ordnet das invertierte Korpus jedem Attributwert die Liste aller Graphknoten zu, in denen dieser Wert verwendet wird. Da Graph**knoten**mengen sehr groß werden können, wird hier stattdessen eine Liste von Korpus**graphen** (repräsentiert durch ihre Nummer) angelegt, die mindest einen Knoten umfassen, der diesen Attributwert verwendet.

Ein Problem für das invertierte Korpus für Attributwerte besteht darin, dass es viele Attribut-Wert-Paare gibt, die in fast jedem Korpusgraph auftreten. Beispiele für das Negra-Korpus sind: `cat="S"`, `pos="NN"` oder `word="."`. Das Ablegen aller Graphen, in denen ein solcher Wert auftritt, ist damit nicht nur redundant, sondern führt zu einem hohen Speicheraufwand. Aus diesem Grund werden nur diejenigen Attributwerte berücksichtigt, deren relative Auftretenshäufigkeit unter einer vorgegebenen Schranke liegt. Die Wahl dieser Schranke stellt einen Kompromiss zwischen Speicherbedarf auf der einen und Menge an relevanter Korpusinformation auf der anderen Seite dar. Der derzeit in der Implementation verwendete Wert beträgt 50%.

Das im Index abgelegte invertierte Korpus strukturiert sich wie folgt (vgl. Abb. 13.2): Zu jedem Attribut des Korpus, und zwar auch zu den ggf. zusätzlich erzeugten Attributen *edge* und *secedge* (vgl. Unterabschnitt 11.1.3 und Unterabschnitt 11.4.3), wird ein eigenes invertiertes Korpus angelegt. In einer Verweisliste wird jedem Attributwert (repräsentiert durch seine Attributnummer) ein Verweis auf eine Liste von Korpusgraphen zugeordnet, die diesen Attributwert für mindestens einen Knoten enthalten. Diese Graphen werden durch ihre Graphnummern repräsentiert. Die Anzahl der Graphknoten bzw. die Länge der zum Attribut gehörenden Graphnummernliste ergibt sich aus der Differenz des nachfolgenden und des aktuellen Verweises.

Für den Fall, dass die Anzahl der Korpusgraphen die Schranke überschreitet, wird eine leere Liste repräsentiert, indem der nachfolgende Verweis mit dem aktuellen übereinstimmt und damit die Anzahl 0 von Graphen kodiert (vgl. Attributwert mit der Nummer 1 in Abb. 13.2). An dieser Stelle mag ein Konflikt vermutet werden: Denn ist ein Attributwert zwar deklariert, tritt er aber im Korpus nicht auf, so ist seine Häufigkeit ebenfalls 0 und würde in genau derselben Weise im invertierten Korpus repräsentiert. Da aber im Attributregister nur die

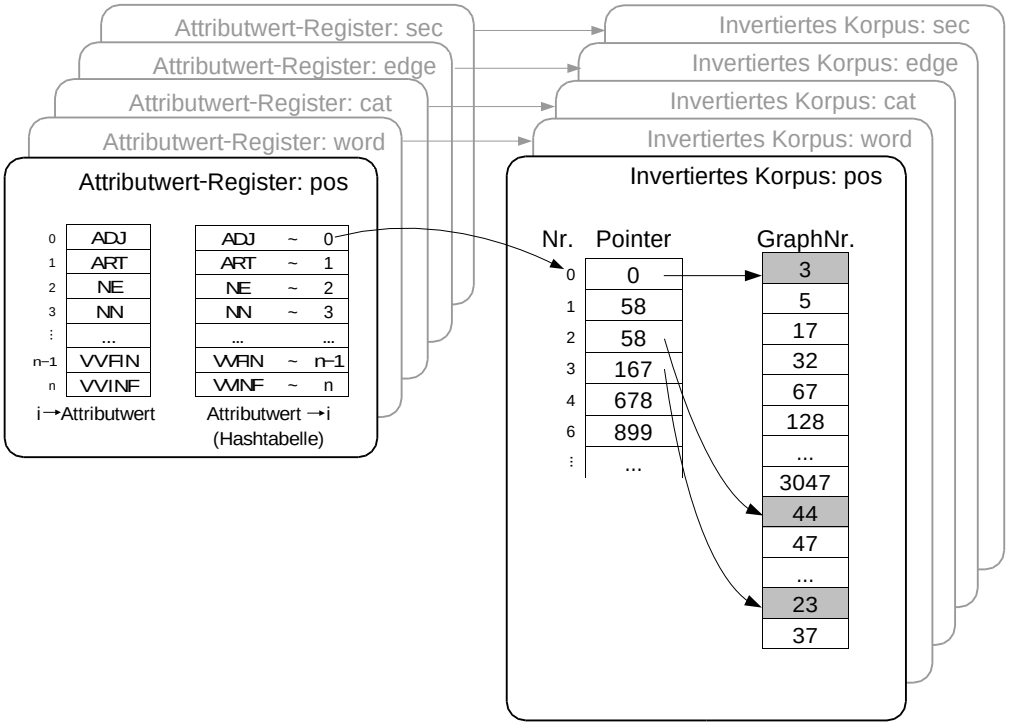


Abbildung 13.2: Repräsentation der invertierten Korpusdaten

tatsächlich verwendeten Attributwerte (d.h. Attributwerte mit einer Mindesthäufigkeit von 1) gespeichert werden, tritt dieser Fall nicht auf.

Ein erster Ansatzpunkt für die Anwendung von Kompressionsverfahren liegt in der Verweisliste, die aufsteigend angeordnet ist. Eine Differenzenkodierung kann hier den Speicherbedarf verringern. Von besonderer Bedeutung ist jedoch die Graphenliste, die im Vergleich zur Verweisliste enorm große Ausmaße annehmen kann (Verhältnis ca. 1:100). Diese Liste ist natürlich nicht aufsteigend angeordnet, da für jeden Attributwert die Graphennummerierung wieder von vorn beginnt. Doch ist jeder Block für sich angeordnet, was durch eine blockweise Differenzenkodierung ausgenutzt werden kann. Dieser Kompressionsansatz ist bereits implementiert worden. Er hat zu einer Reduktion der durchschnittlichen Listengröße auf etwa 25% der Originalgröße geführt.

Das invertierte Korpus stellt nun folgende Indexoperationen zur Verfügung:

- `boolean isInvertedCorpusList(String fname, int valuenumber)`

  Überprüft, ob zum gegebenen Attributwert mit der Nummer *valuenumber* für das Attribut mit dem Namen *fname* eine Liste von Korpusgraphen vorhanden ist. Falls nicht, so hat die Häufigkeit des Attributwerts die vorgegebene Schranke überschritten.

- `int[] getInvertedCorpusList(String fname, int valuenumber)`

Bestimmt zum gegebenen Attributwert mit der Nummer *valuenumber* für das Attribut mit dem Namen *fname* die Liste der Korpusgraphen, in denen dieser Attributwert für mindestens einen Knoten auftritt.

- `int getInvertedCorpusLength(String fname, int valuenumber)`

  Bestimmt zum gegebenen Attributwert mit der Nummer *valuenumber* für das Attribut mit dem Namen *fname* die Länge der Liste der Korpusgraphen, in denen dieser Attributwert für mindestens einen Knoten auftritt. Diese Länge gibt in Relation zur Zahl aller Korpusgraphen die relative (Korpusgraph-)Häufigkeit des Attributwerts an.

Die naheliegende Vorgehensweises bei der Konstruktion des invertierten Korpus lautet: Sammle für jedes Attribut und für jeden Attributwert die Nummern aller Korpusgraphen auf, in denen es mindestens einen Knoten gibt, bei dem dieses Attribut-Wert-Paar vorliegt.

Bei der technischen Realisierung gibt es dabei Probleme mit dem Speicherbedarf, da dieser linear mit der Anzahl der verarbeiteten Korpusgraphen anwächst. Deshalb werden die Daten stets für einen Block von Graphen gesammelt, dessen Größe vorher festgelegt worden ist (z.B. 1.000 Graphen). Die gesammelten Daten werden für einen solchen Block auf die Festplatte geschrieben und der Speicher anschließend wieder freigegeben.

Am Ende der Indexierung werden die Datenblöcke dann paarweise verschmolzen. Da zwei zusammengehörige Listen (z.B. zwei `pos="ITJ"`-Listen) in zwei nacheinander angelegten Datenblöcken aufeinanderfolgende Graphnummern enthalten, besteht der Verschmelzungsprozess aus einer Konkatenation von Listen. In jedem Verschmelzungsschritt halbiert sich die Anzahl der Datenblöcke, bis am Ende ein Gesamtblock übrig bleibt. Der Verschmelzungsalgorithmus wird in (Baeza-Yates und Ribeiro-Neto 1999, Kapitel 8.2) ausführlich beschrieben.

Nach der Verschmelzung werden dann noch diejenigen Attributwerte aussortiert, deren (Korpusgraph-)Häufigkeit über der Häufigkeitsschranke liegt. Die Verweise auf die Graphlisten im Index errechnen sich dann aus den Längen der tatsächlich verwendeten Korpusgraphlisten.

Man beachte, dass die Daten des invertierten Korpus nicht wie die Korpusdaten blockweise portioniert werden können, um den Speicherbedarf durch blockweises Einlesen in den Hauptspeicher gering zu halten (vgl. Abschnitt 11.6). Da die Korpusgraphlisten blockweise nach den Attributwerten angeordnet sind, ist eine Stückelung nach Graphnummern nicht möglich. Hier muss nach alternativen Lösungen gesucht werden.

Der Filteralgorithmus

Voraussetzung für die Anwendung des Filteralgorithmus ist die Disjunktive Normalform. Der Knotenfilter-Algorithmus konzentriert sich auf die Knotenbe-

schreibungen eines Disjunkts. Nach der Erzeugung der Halde sind die in einer Anfrage verwendeten Knotenbeschreibungen in der Knotenbeschreibungstabelle ablesbar. Es werden natürlich nur diejenigen Knotenbeschreibungen berücksichtigt, die auch an eine Attributspezifikation gebunden sind - ungebundene Variablen stellen keinerlei restriktive Information zur Verfügung.

Der Filteralgorithmus untersucht nun alle in Frage kommenden Knotenbeschreibungen auf ihre Restriktivität und filtert im positiven Falle die Menge der Korpusgraphkandidaten aus. Dazu werden folgende Verarbeitungsebenen durchlaufen:

- Eine Anfrage ist genau dann restriktiv, wenn mindestens eine Knotenbeschreibung restriktiv ist. Sind mehrere Knoten restriktiv, so werden die Mengen von Nummern der Korpusgraphen, die die einzelnen Knotenbeschreibungen erfüllen, geschnitten. Gibt es keinen einzigen restriktiven Knoten, so ist die Filterung erfolglos geblieben.

- Eine Knotenbeschreibung ist genau dann restriktiv, wenn mindestens ein Attribut-Wert-Paar der Attributspezifikation restriktiv ist. Gibt es mehr als nur ein restriktives Attribut-Wert-Paar, so werden die Knotengraphen-Mengen, die die Attribut-Wert-Paare erfüllen, geschnitten.

  Da ein evtl. eingehende Kantenbeschriftung intern als *edge*-Attribut verarbeitet wird, können die meist recht aussagekräftigen Kantenbeschriftungen mit berücksichtigt werden. Sekundäre Kanten werden ignoriert, ihre Kantenbeschriftungen werden aber zur späteren Verwendung durch weitere Filteransätze mit im invertierten Korpus abgelegt.

- Ein Attribut-Wert-Paar einer Attributspezifikation ist genau dann restriktiv, wenn der Attributwert aus einer Konstante besteht und die relative Korpusgraphenhäufigkeit dieser Attributbelegung unter der vorgegebenen Schranke von 50% liegt. Dazu wird die Indexoperation `boolean isInvertedCorpusList(String fname, int valuenumber)` verwendet. Ansonsten ist dieses Attribut-Wert-Paar nicht restriktiv.

  Ist der Attributwert erst gar nicht im Attributregister verzeichnet, so tritt er im Korpus nicht auf. Dies ist der denkbar günstigste Fall, denn nun kann die Knotenbeschreibung und damit durch den konjunktiven Kontext das gesamte Anfragedisjunkt niemals erfüllt werden. Der Knotenfilter kann damit innerhalb kürzester Zeit auf Anfragen reagieren, die durch ihre Formulierung niemals einen Korpusgraphen matchen können.

  Für ein restriktives Attribut-Wert-Paar wird die Liste der erfüllenden Korpusgraph-Nummern mit Hilfe der Operation `int[] getInvertedCorpusList()` im erweiterten Index nachgeschlagen und zur Filterung verwendet.

Die Laufzeit des Algorithmus beschränkt sich selbst im schlechtesten Falle auf den Durchlauf durch alle Bedingungen der Anfrage. Da die Korpusgraph-Listen angeordnet sind, ist auch die Schnitt-Operation in linearer Zeit durchführbar. Die Laufzeit der Filterung ist also als sekundär einzuordnen. Als einziger Nachteil bleibt der Ressourcenbedarf, da das invertierte Korpus trotz umfangreicher Komprimierungen vergleichsweise viel Speicherplatz erfordert (ca. 15% des gesamten Index).

Vorteil des Verfahrens ist die Effizienzverbesserung, falls mindestens eine restriktive Knotenbeschreibung auftritt. Im positiven Falle verbessert sich die Laufzeit drastisch, im negativen Falle bleibt der Filter hingegen wirkungslos. Dabei kann eine Anfrage durchaus restriktiv sein, obwohl nicht ein einziger Knoten restriktive Eigenschaften aufweist. Eine Idee zur Ergänzung des Knotenfilters wird im folgenden Unterabschnitt beschrieben.

13.2.2 Ein Relationsfilter

Eine Knotenrelation innerhalb einer Korpusanfrage kann selbst dann sehr aussagekräftig sein, wenn beide beteiligten Knoten keinerlei Aussagekraft besitzen. Im Negra-Korpus beispielsweise enthalten nur ca. 3.700 der 20.600 Korpusgraphen eine Dominanzrelation der Form `[cat="VP"] > [cat="NP"]`. Der Relationsfilter versucht, diese Information nutzbar zu machen.

Dazu wird die Idee des Korpusfilters auf Knotenrelationen verallgemeinert: Für die beiden Basisrelationen bzw. für alle Knotenpaare, zwischen denen die Basisrelation gilt, werden diejenigen Korpusgraphen ermittelt, die diese Struktur enthalten. Bei einer späteren Anfrage `[cat="VP"] > [cat="NP"]` kann die Suche dann auf die ermittelten Korpusgraphen eingeschränkt werden.

Am Beispiel eines Korpus mit zwei Korpusgraphen und der direkten (unbeschrifteten) Dominanzbeziehung soll die Relationsfilter-Idee im Folgenden illustriert werden. Dazu bestehe das Korpus aus den beiden Graphen, die in Abb. 13.3 abgedruckt sind.

Fasst man die Pfade durch die Korpusgraphen als Text auf, so können sehr effektive Textindexierungstechniken wie die *suffix trees* verwendet werden (Baeza-Yates und Ribeiro-Neto 1999, Abschnitt 8.3). Dazu werden die Kategorienbelegungen eines ausgewählten Attributs (im nachfolgenden Beispiel des Attributs *cat*) für alle Pfaddurchläufe durch den Korpusgraphen, beginnend an einem beliebigen Graphknoten, als Kantenbeschriftungen eines Tries notiert. Für eine Einführung in die Trie-Datenstruktur sei auf (Ottmann und Widmayer 1993, Seite 400 ff.) verwiesen. An jedem Knoten des Tries werden die Graphnummern derjenigen Korpusgraphen notiert, die die durchlaufende Struktur enthalten. Der für das Beispielkorpus entstehende Trie ist in Abb. 13.4 abgebildet.

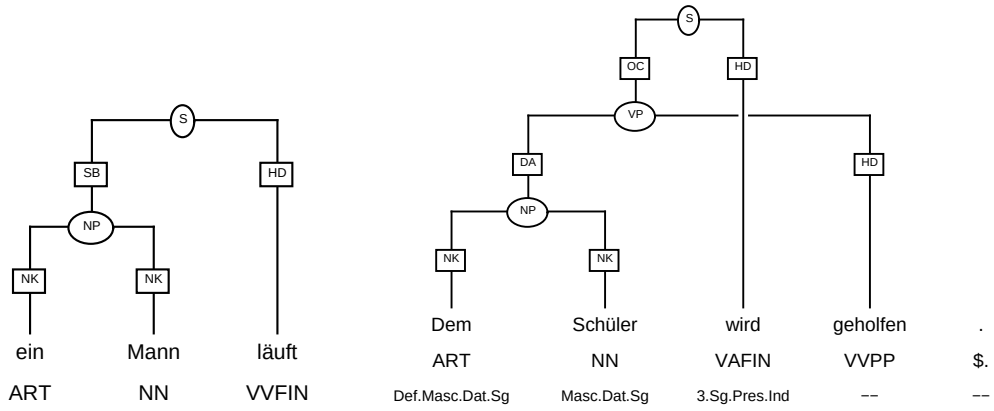


Abbildung 13.3: Ein Beispielkorpus bestehend aus zwei Graphen

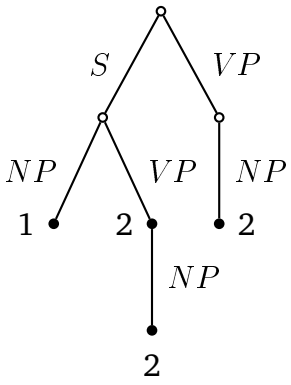


Abbildung 13.4: Ein Trie zur Kodierung der Korpusdominanzen

Auf der ersten Ebene des Tries werden noch keine Graphnummern-Listen verwaltet, da die dort kodierbare Information, welche Korpusgraphen einen Knoten mit dem angegebenen Attributwert aufweisen, bereits im invertierten Korpus abgelegt ist.

Mit Hilfe dieses Tries können nun beliebige Dominanzrelationen der Anfrage zur Filterung verwendet werden. Dazu wird die Relation in einen regulären Ausdruck überführt, der durch den Trie effizient verarbeitet werden kann. Die Attributwerte bilden dabei die Konstanten des verwendeten Alphabets.

Dazu einige Beispiele:

- `[cat="VP"] > [cat="NP"]` $\rightarrow$ `(VP) (NP)`

  Suche beginnend mit dem Wurzelknoten einen Trie-Pfad der Länge 1 von der Form `VP NP`. Im Beispiel ist am Ende des Pfades abzulesen, dass nur der Korpusgraph mit der Nummer 2 in Frage kommt.

- `[cat="S"] >2 [cat="NP"]` → `(VP) . (NP)`

  Suche einen Pfad, der mit `VP` beginnt, dann genau einen beliebigen Übergang aufweist und anschließend mit einem `NP`-Übergang abschließt. Im Beispiel kommt nach dem Trie nur der Graph mit der Nummer 2 in Frage.

- `[cat="S"] >* [cat="NP"]` → `(S) .* (NP)`

  Suche einen Pfad beliebiger Länge, der mit `S` beginnt und auf `NP` endet. Im Beispiel kommen hier beide Graphen in Frage.

Eine speichersparende und trotzdem effizient zu verarbeitende Variante des *suffix trees* stellen die sogenannten *PAT trees* dar (vgl. Baeza-Yates und Ribeiro-Neto 1999, Kapitel 8.3 und Baeza-Yates und Gonnet 1996). Sie vermeiden unäre Kantenübergänge, die die Höhe des Suchbaumes unnötig ausdehnen und den Suchraum aufblähen. In der vorliegenden Verwendung wird beispielsweise die Auswertungseffizienz einer allgemeinen Dominanz durch die Höhe des Baums maßgeblich beeinflusst.

Ausgeklammert ist in diesem Beispiel die Eigenschaft des Datenmodells, dass statt einem einzigen Attribut-Wert-Paar an jedem Graphknoten eine Liste von Attribut-Wert-Paaren sowie eine Kantenbeschriftung vorliegen kann. So fehlen in diesem Beispiel die Terminalknoten, die völlig andere Attribute aufweisen können. Als Lösung könnte jeder Knoten des Tries aus einer Liste aller Attribute und ihrer ggf. vorliegenden Attributwerte bestehen. Damit wächst aber die Anzahl der Verzweigungen im schlechtesten Fall exponentiell mit der Anzahl der Attribute. Eine alternative Vorgehensweise besteht darin, für jedes Attribut einen eigenen Trie zu verwalten und diese Attribut-Tries beim Filtern zu kombinieren.

Dieses Problem erschwert in besonderer Weise die Übertragung der Filteridee auf die Filterung von Präzedenzrelationen. Denn hier müssen in jedem Falle Terminal- und Nichtterminal-Knoten mit ihren unterschiedlichen Attributen betrachtet werden. Als Ausweg könnten beispielsweise nur Präzedenzrelationen über Terminalknoten ausgewertet werden. Diese Einschränkung ist sinnvoll, da die Anwender erfahrungsgemäß eher selten von der Präzedenzrelation zwischen beliebigen Knoten Gebrauch machen.

Da ein Relationsfilter eine ideale Ergänzung des Knotenfilters darstellt, wird die Relationsfilter-Idee als einer der nächsten Schritte realisiert werden.

13.3 Anfrageoptimierung

13.3.1 Einführung

Die Integration von Optimierungsstrategien in die Architektur eines anfrageverarbeitenden Systems kann nach (Vossen 1994, Kapitel 17) wie folgt beschrie-

ben werden: Anfragen an das System werden in einer sogenannten Hochsprache formuliert, d.h. in einer formalen Sprache, dessen Design sich stärker an der Bedürfnissen der Anwender als an der leichten Verarbeitbarkeit durch die Maschine orientiert. Die in der Hochsprache formulierte Anfrage wird in eine Folge von Operationen übersetzt, die vom Auswertungsprozessor ausgeführt wird. Die Aufgabe der Anfrage-Optimierung besteht darin, eine Auswertungsstrategie zu entwickeln, die für eine gegebene Anfrage durch geschickte Anordnung der Operationsfolge die meisten Systemparameter optimiert: verbrauchte Zeit, benötigter Hauptspeicher usw. Da die Berechung der optimalen Strategie meist ein sehr komplexes Problem darstellt, besteht die Zielsetzung der Optimierung nicht unbedingt darin, die beste Strategie zu finden, sondern eine schlechte Strategie zu vermeiden.

Zur groben Klassifizierung von Optimierungsansätzen werden *High-Level*-Optimierung und *Low-Level*-Optimierung unterschieden. Die High-Level-Optimierung wendet syntaktische Umformungsregeln der Hochsprache an (z.B. Kommutativität von Operationen), die unabhängig von der Implementation sind. Die Low-Level-Optimierung bindet dagegen die Eigenschaften und Parameter der tatsächlichen Implementation mit in die Optimierungsstrategie ein.

Die umfangreichsten Arbeiten auf dem Gebiet der Anfrageoptimierung sind im Datenbanken-Umfeld entwickelt worden, sei es für relationale Datenbanken (für eine Übersicht siehe Vossen 1994, Kapitel 17) oder objektorientierte Datenbanken (für eine Übersicht siehe Yu und Meng 1998, Kapitel 2). Für relationale Datenbanken besteht ein Hauptansatzpunkt auf High-Level-Ebene in der Anwendung von Umformungsregeln auf die in der Hochsprache (meist SQL) formulierten Anfrage bzw. in der Anwendung von Rechenregeln auf die algebraische Anfragerepräsentation. Auf Low-Level-Ebene konzentrieren sich die Ansätze u.a. auf die interne Realisierung der Verbund-Operation.

Daneben gibt es eine Reihe von Arbeiten zur Optimierung der Verarbeitung semistrukturierter Daten. Eine Übersicht über das Forschungsfeld semistrukturierter Daten geben Abiteboul et al. (2000). Für die Arbeiten zur Anfrageoptimierung seien hier stellvetretend die Arbeiten zum *Lore*-Projekt genannt (vgl. McHugh und Widom 1999 und McHugh 2000). Hier konzentrieren sich die Ansätze meist darauf, den Auswertungsplan in eine möglichst günstige Reihenfolge zu bringen.

Auch wenn diese Arbeiten auf Datenmodellen arbeiten, die vom vorliegenden Datenmodell verschieden sind, so ist doch eine zentrale Idee allen verfolgten Ansätzen gemein: Die Folge der Operationen, die hier der Reihenfolge der Anwendung von Kalkülregeln entspricht, wird in eine möglichst geschickte Reihenfolge gebracht.

Auf der Ebene der Anfrageoptimierung befasst sich die Realisierung dieser Idee in den Unterabschnitten 13.3.2 und 13.3.3 mit der Anordnung des Anfrageskeletts und der Anordnung der formulierten Bedingungen in der Halde. Da lediglich Umformungsregeln des Kalküls verwendet werden, bleibt die Se-

mantik der Anfrage unangetastet. Dadurch ist die Korrektheit und Vollständigkeit der erweiterten Implementation sichergestellt. Da bei der Entwicklung der Anordnungsstrategie stets auch die Eigenschaften der internen Anfrageverarbeitung berücksichtigt wurden, stellt diese Herangehensweise eine Low-Level-Optimierung dar.

Daneben wird in Unterabschnitt 13.3.4 als weitere Optimierung die Reduzierung der Knotenkandidatenmengen fokussiert, die bei der Realisierung des nicht-deterministischen Knotentests eine Schwachstelle darstellen.

13.3.2 Optimierung der Halde

Während Knotenbeschreibungen nur eine Attributspezifikationsvariable und eine evtl. Zuordnung zu einer Knoten-ID umfassen, bestehen auf Attributspezifikations-Ebene (bedingt durch die konjunktive Aufzählung von Attribut-Wert-Paaren) und auf Attributwert-Ebene (bedingt durch die Aufzählung von Disjunkten) Ansatzmöglichkeiten zur Veränderung der Verarbeitungsreihenfolge und damit Ansatzpunkte zur Effizienzsteigerung. Die Konzepte und Implementationen für diese beiden Ebenen werden im Folgenden beschrieben.

Attributspezifikationen

Bei der Einweisung einer Knotenbeschreibung in die Halde werden die Attribut-Wert-Paare nach dem Algorithmus zur Erzeugung der Halde (vgl. Unterabschnitt 12.3.2) in ihrer ursprünglichen Reihenfolge in die Attributspezifikation übertragen. Die Reihenfolge der Paare kann dabei die Effizienz der Verarbeitung maßgeblich beeinflussen. Dies wird an folgendem Beispiel deutlich:

```
[pos=("NN"|"NE") & word="Haus"]
```

Wird diese Anfrage auf dem Negra-Korpus ausgeführt, so wird für jeden Knotenkandidaten zunächst der Wortart-Test durchgeführt. Dieser Test ist für fast 93.000 Knoten des Korpus erfolgreich. Erst bei erfolgreichem Test kommt es zum Test der Wortform. Die Wortform *Haus* liegt aber im Korpus nur bei 88 Knoten vor. Insgesamt wird also in 93.000 Fällen auch der zweite Vergleich durchgeführt, von denen aber nur 88 Fälle erfolgreich sind. Werden nun die beiden Attribute vertauscht, d.h. wird der Wortform-Test vorgezogen, sieht das Bild anders aus. Nur in 88 Fällen kommt es überhaupt zu einem zweiten Vergleich, fast 93.000 zweite Vergleiche werden eingespart.

Bei dieser Rechnung sollte berücksichtigt werden, dass die Ersparnis für eine solche Attributspezifikation im Rahmen einer realen Anfrage geringer ausfällt. Denn in vielen Fällen wird der Backtracking-Algorithmus erst gar nicht bis zum Binden dieser Knotenbeschreibung gelangen.

Die hier beschriebene Heuristik sieht vor, selten matchende Attribute mit zugleich leicht zu evaluierenden Attributwerten nach vorne zu schieben, um weitere Vergleiche einzusparen. Denn während bei einem erfolgreichen Test gegen einen Korpusknoten ohnehin alle Attribute der Reihe nach bearbeitet werden, sollte der erfolglose Test an einem möglichst frühen Attribut scheitern.

Dazu wird in der Implementation eine Kostenfunktion verwendet, die jedem Attribut seine geschätzten Kosten (d.h. die Unifikationskosten) zuordnet. Die Attribute werden dann nach ihren Bewertungen angeordnet. Sei a das zu bewertende Attribut und $range(a)$ die Anzahl aller im Korpus auftretenden konstanten Belegungen von a. Diese Anzahl ist aus dem Attributregister von a im Index zu entnehmen (vgl. Unterabschnitt 11.1.1). Sei zudem d der zu a abgelegte Attributwert. Dann wird die Bewertung von a definiert als:

$$
\begin{aligned}
k(a) &= k(d) + \frac{1}{range(a)} \\
k(d) &= k\left((d_{1,1}\& \ldots \& d_{1,n_1}) | \cdots | (d_{m,1}\& \ldots \& d_{m,n_m})\right) \\
&= \sum_{i=1}^{m} \sum_{j=1}^{n} k(d_{i,j}) \\
k(d_{i,j}) &= \begin{cases} 1 & \text{, falls } d_{i,j} \text{ Konstante oder Typ} \\ 3 & \text{, falls } d_{i,j} \text{ negierte Konstante oder negierter Typ} \end{cases}
\end{aligned}
$$

Die Kosten eines Attributs werden geschätzt als die Kosten, die bei der Unifikation mit einem Attribut-Wert-Paar eines Korpusknotens entstehen. Da in vielen Anfragen nur eine Konstante als Attributwert vorliegt (z.B. in `[pos="NN" & word="Haus"]`) und die Bewertung in beiden Fällen 1 ergibt, gibt hier die Kardinalität der Menge aller Attributwerte den Ausschlag. Denn für ein Attribut mit vielen Belegungen ist die mittlere relative Häufigkeit eines einzelnen Werts niedriger als für ein Attribut mit nur wenigen Werten. Damit werden seltene Attributbelegungen bevorzugt.

Die beschriebene Optimierungsstrategie ist bereits implementiert worden. Die Ergebnisse zeigen Effizienzsteigerungen um bis zu 50%. Im Falle der Beispielanfrage `[pos=("NN"|"NE") & word="Haus"]` reduziert sich die Verarbeitungszeit um 30%.

Attributwerte

Die beschriebene Idee zur Anordnung der Attribut-Wert-Paare lässt sich analog auf die Anordnung der Bestandteile jedes Attributwerts übertragen. Attributwerte werden auf der Halde in Disjunktiver Normalform verwaltet. Da die Konjunkte eines Disjunkts durch das Design der Anfragesprache nur wenige Ausdrucksmöglichkeiten bieten, die nicht unmittelbar zu einem Clash oder einer Reduktion führen, umfassen die Disjunkte in der Regel nur wenige Konjunkte,

in den allermeisten Fällen genau ein Konjunkt. Daher lohnt sich der Aufwand einer Anordnung der Konjunkte nicht.

Die Anordnung der Disjunkte hingegen beeinflusst die Verarbeitungsgeschwindigkeit deutlich und wird deshalb im Folgenden diskutiert. Als Ausgangsbeispiel dient die folgende Anfrage, die wieder auf dem Negra-Korpus verarbeitet werden soll:

```
[pos=("ITJ"|"NN")]
```

Während *ITJ* eine seltene Wortart darstellt (Interjektion, nur 25 Vorkommen), ist die Wortart *NN* sehr häufig (Nomen, fast 74.000 Vorkommen). Der Ausdruck `[pos=("ITJ"|"NN")]` wird also von rund 74.000 Knoten erfüllt.

Wenn im Folgenden die mögliche Ersparnis durch die Umordnung des Attributwerts diskutiert wird, muss stets berücksichtigt werden, dass es sich hier um einen konstruierten Fall handelt. Im Kontext einer komplexen Anfrage wird die formulierte Knotenbeschreibung in vielen Fällen gar nicht gebunden, da der Backtracking-Algorithmus bereits in einer vorausgehenden Stufe gescheitert ist. Auch darf nicht vergessen werden, dass bei allen Knotenkandidaten, die weder *NN*- noch *ITJ*-Knoten sind, die Reihenfolge der Konstanten keinen Einfluss auf die Anzahl der Vergleiche hat. D.h. nur im Falle eines erfolgreichen Knotentests findet überhaupt eine Ersparnis statt. Unter der Annahme, dass alle terminalen Knoten den Knotentest durchlaufen, handelt es sich im Falle des Negra-Korpus aber immerhin um 281.000 Knoten, die beide Vergleiche erfordern.

Insgesamt ist die Ersparnis lohnenswert: Da die Unifikation eines Korpusknotens mit der Attributspezifikation der Knotenbeschreibung aus der Anfrage für einen positiven Attributwert-Abgleich mindestens einen Vergleich erfordert, finden hier fast 74.000 Vergleiche von *NN*-Graphknoten mit der *ITJ*-Konstante statt, die erfolglos sind und erst im zweiten Vergleich mit der *NN*-Konstante zum Erfolg führen. Wird die Reihenfolge der Konstanten dagagen umgedreht, kommt es in nur 25 Fällen zu einem zweiten erfolgreichen Vergleich. Dagegen werden fast 74.000 Vergleiche eingespart, da die *NN*-Graphknoten schon beim ersten Vergleich erfolgreich sind.

Die beschriebene Heuristik sieht also vor, die häufigen und zugleich schnell auszuwertenden Disjunkte nach vorn zu schieben. Denn während bei einer erfolglosen Unifikation ohnehin alle Konjunkte der Reihe nach verarbeitet werden, kann so zumindest eine erfolgreiche Unifikation möglichst schnell abgeschlossen werden.

Dazu werden die Auswertungskosten aller Disjunkte eines Attributwerts abgeschätzt und die Disjunkte nach ihrer Kostenbewertung angeordnet. Sei nun d ein Disjunkt und $rel(c)$ sei die relative Graphhäufigkeit einer Attributwert-Konstante, die aus dem erweiterten Index abgelesen wird (vgl. Unterabschnitt 13.2.1). Dann wird die Bewertung von d definiert als:

$$k(d) \quad = \quad k(d_1 \& \ldots \& d_n)$$

$$
\begin{aligned}
& = \sum_{i=1}^{n} k(d_i) \\
k(d_{ij}) & = \begin{cases} 1 \Leftrightarrow rel(d_i) & \text{, falls } d_i \text{ Konstante} \\ 1 & \text{, falls } d_i \text{ Typ} \\ 2 & \text{, falls } d_i \text{ negierte Konstante oder Typ} \end{cases}
\end{aligned}
$$

Die Kosten eines Attributwerts werden also als die Kosten geschätzt, die bei der Unifikation mit einer Attributwert-Konstante entstehen. Da in vielen Fällen nur Konstanten als Disjunkte des Attributwerts vorliegen, gibt hier die relative Häufigkeit des Auftretens den Ausschlag. Diejenige Konstante mit der höchsten relativen Häufigkeit wird bevorzugt.

Auch diese Optimierungsstrategie ist bereits implementiert worden. Wie bereits am Beispiel angedeutet, werden durch diese Heuristik zwar viele Vergleiche eingespart, die Zahl der gesparten Vergleiche macht aber im Verhältnis zur Zahl aller Vergleiche nur einen geringen Teil aus. Die Effizienzsteigerungen liegen tatsächlich im Rahmen von bis zu 10% und fallen damit eher moderat aus. Im Anfragebeispiel `[pos=("ITJ"|"NN")]` wird die Verarbeitungsdauer um etwa 5% reduziert.

13.3.3 Optimierung des Anfrageskeletts

Da der Test eines Korpusgraphen gegen eine Anfrage, bestehend aus dem Anfrageskelett und der Halde, das Anfrageskelett von links nach rechts bearbeitet, kann auch hier eine Anordnung der Reihenfolge die Verarbeitungsgeschwindigkeit beeinflussen. Da der Aufwand zum Testen eines Knotens um ein Vielfaches höher liegt als der Aufwand für einen Relations- oder Prädikatstest, bringt jeder eingesparte Knotentest einen hohen Zugewinn.

Zur Illustration soll eine Anfrage analysiert werden, die eine Nominalphrase beschreibt, die aus einem Artikel, einem Adjektiv und einem Nomen besteht. Die Nominalphrase soll genau diese drei Token in der angegebenen Reihenfolge umfassen. Viele Anwender werden diese Anfrage auf die folgende Art und Weise formulieren:

```
#np:[cat="NP"] &
#art:[pos="ART"] &
#adj:[pos="ADJA"] &
#noun:[pos="NN"] &
#np > #art &
#np > #adj &
#np > #noun &
#art . #adj &
#adj . #noun &
arity(#np,3)
```

Die Anfrageformulierung des Anwenders ist sehr übersichtlich und gut strukturiert. Zuerst werden die Knotenbeschreibungen spezifiziert und anschließend diese Knoten zu einem Graphfragment zusammengesetzt. Durch die Erzeugung der Halde zerfällt diese Anfrage in ein Anfrageskelett. Sie hat dann die folgende Form:

```
#np & #art & #adj & #noun &
(#np > #art) & (#np > #adj) & (#np > #noun) &
(#art . #adj) & (#adj . #noun) &
arity(#np,3)
```

Leider ist die Reihenfolge der Konjunkte die denkbar ungünstigste. Grund dafür ist die Vorgehensweise des Test-Algorithmus. Er bindet bei der vorliegenden Reihenfolge zunächst alle vier Knotenbeschreibungen. Wenn es in einem Korpusgraph K Knoten gibt, so werden K^4 Knoten gebunden.

Doch bereits durch eine kleine Änderung kann die Anfrageverarbeitung drastisch beschleunigt werden. Dazu wird die Dominanzrelation `#np > #art`, die die Beziehung zwischen den beiden erstgenannten Knoten definiert, direkt hinter die beiden Knotendefinitionen geschoben:

```
#np & #art & (#np > #art) &
#adj & #noun &
(#np > #adj) & (#np > #noun) &
(#art . #adj) & (#adj . #noun) &
arity(#np,3)
```

Der Effekt dieser Operation besteht darin, dass zwar noch immer bis zu K^2 Knotenpaare gebunden werden. Die nachfolgende Relation `#np > #art` schließt aber für sehr viele dieser Paare die Weiterverarbeitung aus. Im Falle des Negra-Korpus gibt es beispielsweise 78.000 *NP*-Knoten und 71.000 *ART*-Knoten, d.h. ca. 500.000 `NP-ART`-Paare, aber nur 50.000 *NP*-Knoten, die einen *ART*-Knoten dominieren. Tatsächlich wird die Verarbeitungsgeschwindigkeit gegenüber der ursprünglichen Variante in etwa verfünffacht.

Die optimale Anordnung der Anfrage ist damit noch nicht erreicht. Die folgende Anordnung kann die Anfrageverarbeitung noch einmal vervierfachen, wodurch sich insgesamt eine Verzwanzigfachung (!) der Geschwindigkeit ergibt:

```
#np & arity(#np,3) & #adj & (#np > #adj) &
#art & (#np > #art) &  (#art . #adj) &
#noun & (#np > #noun) & (#adj . #noun)
```

Als interessante Beobachtung kann dabei festgehalten werden, dass Prädikate eine Sonderstellung genießen. Sie binden gegenüber einer Knotenrelation

nur einen Knoten und ihr Test erfolgt in deutlich kürzerer Zeit. Sie können daneben sehr restriktiv sein (z.B. eine vorgegebene Token-Stelligkeit) und blocken viele Verarbeitungspfade bereits zu einem frühen Zeitpunkt ab. Diese Feststellung ist zentraler Ansatzpunkt für die folgende Heuristik.

Die bislang informell beschriebene Relationsfilter-Idee wird zunächst spezifiziert:

- In einer Anfrage gebe es k Konjunkte. Es kann sich dabei um Knotenbeschreibungen, Prädikate und Knotenrelationen handeln.
- Zulässige Anordnungen der Anfrage sind all diejenigen Permutationen der k Konjunkte, die folgende Auflage erfüllen: Kommt eine Knotenbeschreibungsvariable `#v` bzw. `#w` innerhalb einer Knotenrelation `#v R #w` oder einem Prädikat `p(#v)` vor, und ist diese Variable keine ungebundene Variable, so muss die zugehörige Knotenbeschreibung von `#v` bzw. `#w` vor ihrer ersten Verwendung in einer Knotenrelation bzw. in einem Prädikat stehen.

  Beispiel: In der Anfrage `#v:[cat="NP"] & arity(#v,2)` darf das Prädikat nicht vor die Knotenbeschreibung geschoben werden. Die Knotenbeschreibung dient im Verarbeitungsalgorithmus als Produzent zur Erzeugung aller Knotenkandidaten von `#v` (vgl. Unterabschnitt 12.2.1).
- Gesucht ist ein Verfahren, das für alle zulässigen Anordnungen diejenige findet, die die geringsten Auswertungskosten verursacht.

Bedingt durch die hohe Komplexität dieses Problems ist es nicht praktikabel, wirklich alle zulässigen Anordnungen zu bilden und einzeln zu bewerten. Stattdessen ist eine Heuristik gefragt, die eine sinnvolle Anordnung bestimmt, die der optimalen möglichst nahe kommt.

An dieser Stelle kommt die Beobachtung zum Tragen, dass Prädikaten eine Sonderstellung zukommt. Als Heuristik wird die Anfrage auf folgende Art und Weise umgeordnet: An den Anfang kommen alle in der Anfrage verwendeten Prädikate. Ihre genaue Anordnung richtet sich nach einer Bewertungsfunktion (s.u.). Es folgen die Knotenrelationen, die ebenfalls nach einer Bewertungsfunktion angeordnet werden. Den Abschluss bilden die Knotenbeschreibungen. Durch diese Konvention befindet sich die Anfrage anschließend nicht mehr in der durch die Pre-Normalisierung vorgegebenen Form (vgl. Unterabschnitt 12.2.1). D.h. der Prenormalisierungsschritt muss ein weiteres Mal durchgeführt werden.

Die Anordnung der Prädikate richtet sich nach einer Bewertungsfunktion, die die Restriktivität des Prädikats schätzen soll. Diese Restriktivität setzt sich zusammen aus dem vorliegenden Prädikatstyp und evtl. Parameter und der Restriktivität der an das Prädikat gebundenen Knotenbeschreibung. Die Restriktivität einer Knotenbeschreibung ist aber nur schwer zu bewerten: Wird

eine schnell zu testende Attribut-Spezifikation bevorzugt, ist sie meist nur wenig restriktiv. Eine restriktive Attribut-Spezifikation hingegen umfasst sehr viele Attribut-Wert-Paare, was die Auswertung verlangsamt. Aus diesem Grunde lässt die implementierte Heuristik die gebundene Knotenbeschreibung außer Acht.

Als primäres Bewertungskriterium wird stattdessen die Häufigkeit der gebundenen Knotenbeschreibungsvariable in allen Prädikaten betrachtet. Denn ist eine Variable `#v` an mehr als nur ein Prädikat gebunden, findet bei der Auswertung des zweiten Prädikats kein Knotentest mehr statt. Die Kombination der beiden Prädikate schränkt damit die verfolgten Pfade frühzeitig ein. Bei übereinstimmender Häufigkeit werden die Prädikate nach ihrem Typ bewertet: Am restriktivsten ist das `root`-Prädikat, gefolgt von `tokenarity` und `arity` mit exakt angegebener Stelligkeit, den beiden Stelligkeitsprädikaten mit Bereichsstelligkeit und schließlich den beiden `(dis)continuous`-Prädikaten.

Zur Bewertung der Knotenrelationen wird eine vergleichbare Strategie verfolgt. Als primäres Kriterium dient die Summe der Häufigkeiten der beiden gebundenen Knotenbeschreibungsvariablen in allen Prädikaten. Da die Prädikate nach vorn geschoben werden, sind die verwendeten Variablen bereits gebunden. Damit sind diejenigen Knotenrelationen zu bevorzugen, die über Prädikatsbindungen verfügen. Bei übereinstimmender Häufigkeit dient als sekundäres Kriterium die Summe der Häufigkeiten der beiden Knotenbeschreibungsvariablen in allen Knotenrelationen. Denn ist analog zur Prädikatsbindung eine Variable `#v` noch an weitere Knotenrelationen gebunden, so ist für die hinteren Relationen der Knotentest von `#v` überflüssig. Im Falle auch hier übereinstimmender Häufigkeit wird schließlich der Relationstyp bewertet: Am restriktivsten sind die n-stufige Dominanz und Präzedenz, es folgen direkte Dominanz und Präzedenz sowie die Geschwisterbeziehung. Bereichsrelationen (z.B. `>m,n`) werden gegenüber allgemeinen Relationen bevorzugt.

Die beschriebene Heuristik ist bereits implementiert worden. Sie führt für die Beispielanfrage zu folgender Anfrageanordnung, die in weiten Teilen mit der manuell ermittelten Anordnung übereinstimmt.

```
#np & arity(#np,3) & #adj & (#np > #adj) &
#art & (#np > #art) &
#noun & (#np > #noun) & (#art . #adj) & (#adj . #noun)
```

Eine Abweichung der Laufzeiten für die intellektuell und maschinell umgeordnete Variante der Beispielanfrage war nicht messbar.

13.3.4 Reduzierung der Knotentest-Kandidatenmenge

Eine entscheidende Stelle für die Effizienz des Verfahrens ist die Knotenauswahl. Soll der Nichtdeterminismus des Kalküls aufgelöst werden, müssen hier

prinzipiell alle Knoten eines Korpusgraphen einzeln gegen eine Knotenbeschreibung getestet werden (vgl. Unterabschnitt 12.4.2). Neben dem Aufwand zur Einweisung in die Halde und Durchführung des Konsistenzchecks kommt die Stack-Verwaltung, die einen hohen Speicherbedarf verursacht. Und dies in vielen Fällen eigentlich unnötig, falls der Knotentest erfolglos bleibt.

In den allermeisten Fällen ist immerhin die Domäne des Knotens bekannt und kann aus der an die Knotenbeschreibung gebundenen Attributspezifikation abgelesen werden. Desweiteren kann die evtl. unbekannte Domäne einer Knotenbeschreibung zuweilen noch vor der Evaluation anhand ihrer Verwendung in einer Knotenrelation abgelesen werden. Der Typ der Knotenvariable `#v` in der Anfrage `#v > [pos="NN"]` beispielsweise ist zwar eigentlich beliebig (d.h. `FREC`). Als Mutterknoten eines anderen Knotens kann `#v` jedoch nur ein Nicht-Terminalknoten sein, d.h. vom Typ `NT`. Diese Information kann bereits im Rahmen der Erzeugung der Halde eingefügt werden.

Es verbleibt dennoch der Test aller Terminal- bzw. Nichtterminal-Knoten eines Korpusgraphen. Um diesen Aufwand zu reduzieren, ist ein Knoten-Schnelltest entwickelt worden. Er unterzieht jeden Knoten noch vor der Einweisung in die Kandidatenmenge einem in sehr kurzer Zeit durchzuführenden Test, ob er überhaupt als Kandidat in Frage kommt. Dieser Test kann damit als ein Filter auf der Ebene der Knotenauswahl charakterisiert werden.

Um die Laufzeit des Tests zu begrenzen, wird noch während der Haldenoptimierung für jede Attributspezifikation dasjenige Attribut att bestimmt, das einen besonders aussagekräftigen Attributwert besitzt. Damit der Schnelltest auch wirklich schnell durchgeführt werden kann, muss es sich beim Wert dieses Attributs um eine Konstante c handeln. Die Bedingung $att = c$ ist damit (durch den konjunktiven Kontext) eine notwendige Bedingung für eine positive Unifikation mit einem Korpusknoten.

Bei mehreren konkurrierenden Attributen entscheidet die niedrigere relative Auftretenshäufigkeit, d.h. die relative Graphhäufigkeit der Vorkommen von c im Korpus, die aus dem erweiterten Index mit Hilfe der Operation `int getInvertedCorpusLength(String fname, int valuenumber)` in Bezug zur Anzahl der Korpusgraphen ermittelt werden kann (vgl. Unterabschnitt 13.2.1). In der Anfrage `[pos="NN" & word="Haus"]` beispielsweise fällt die Wahl auf das *word*-Attribut, da die relative Häufigkeit der Wortform *Haus* wesentlich niedriger ist als die relative Häufigkeit der Wortart *NN* (Nomen).

Soll nun zum Abgleich gegen eine Korpusbeschreibung `#v` für einen Korpusgraphen die Knotenkandidaten-Menge bestimmt werden, wird zunächst jeder prinzipielle Kandidat einem Schnelltest unterzogen: Ist `#f` die Attributspezifikation zur Knotenbeschreibung `#v`, so kann nun der Attributwert für das vorher ermittelte Testattribut att überprüft werden. Dazu wird die Attributbelegung des Kandidaten aus dem Index ausgelesen. Diese Belegung wird mit der Attributwert-Konstante verglichen, die das Attribut att in der Attributspezifikation `#f` aufweist. Bei Nicht-Übereinstimmung ist der Knotentest für den

Graphknoten überflüssig geworden. Der Graphknoten wird nicht in die Kandidatenmenge eingewiesen.

Im Falle der Anfrage `[pos="NN" & word="Haus"]` kommen damit nur diejenigen Graphknoten in die Kandidatenmenge, die als Wert des Attributs *word* die Wortform *Haus* aufweisen. Durch diese Vorgehensweise gelangen für die Beispielanfrage bei Auswertung auf dem Negra-Korpus (88 Vorkommen der Wortform *Haus* in 20.600 Korpusgraphen) durchschnittlich weniger als ein Knoten pro Graph in die Kandidatenmenge. Die Ersparnis ist also immens.

Der Knoten-Schnelltest bringt allerdings nur dann einen Gewinn, wenn die Attributwerte innerhalb der Attributspezifikationen aus Konstanten bestehen. In der Praxis ist dies allerdings in über 90% der formulierten Knotenbeschreibungen der Fall. Erste Messungen haben gezeigt, dass der Knoten-Schnelltest die Auswertung in günstigen Fällen um bis zu 50% beschleunigt.

13.4 Java-spezifische Ansätze

Eine weitere Verbesserung der Laufzeit kann durch das konsequente Ausnutzen der Möglichkeiten der verwendeten Programmiersprache Java erreicht werden. Die im Rahmen dieser Arbeit einsetzbaren Techniken werden in den beiden folgenden Unterabschnitten beschrieben.

13.4.1 Parallelisierung

Durch die Normalisierung der Anfrage in eine Disjunktive Normalform entsteht eine Liste von Disjunkten, die völlig isoliert voneinander verarbeitet werden können. Es empfiehlt sich hier, diese getrennte Verarbeitung durch eigene Prozesse, genauer durch eigene Prozessoren zu realisieren. Java bietet hierzu die Technologie der *Java Threads* an. Für eine umfangreiche Einführung und Diskussion dieser Technologie sei auf Oaks und Wong (1999) verwiesen. Eine Thread-gestützte Implementation wird dann auf einer Mehr-Prozessor-Maschine etwa um den Faktor schneller, der durch die Anzahl der Prozessoren vorgegeben ist.

Falls die Normalisierung zu genau einem Disjunkt führt, bestehen immer noch Paralleliserungsmöglichkeiten. Bedingt durch die graphenweise Evaluation der Anfrage gegen die Korpusgraphen kann die Evaluation des Korpus beispielsweise durch zwei parallele Threads erfolgen, die das Korpus aufsteigend beginnend mit dem ersten bzw. absteigend beginnend mit dem letzten Graphen evaluieren. Sobald sich die beiden Threads treffen, ist die Evaluation abgeschlossen. Diese Vorgehensweise lässt sich natürlich auf beliebig viele Threads ausdehnen.

13.4.2 Eine Client/Server-Architektur

Wenn zwei Anwender zur gleichen Zeit auf demselben Korpus arbeiten, müssen beide das Korpus bzw. Teile des Korpus (in der internen Verarbeitung als Index repräsentiert) im Speicher halten. Dies ist ein Nachteil vor allem bei der Arbeit mit sehr umfangreichen Korpora.

Damit sich die Anwender das Korpus teilen können, wird eine Client/Server--Architektur aufgebaut. Auf einem leistungsstarken Rechner läuft ein zentraler Server-Prozess, der jedes benötigte Korpus genau einmal im Speicher hält. Die Anwender verwenden auf ihren lokalen Rechnern einen Client, der selbst keine Auswertungsfunktionalität mehr umfasst, sondern lediglich für die Eingabe von Anfragen und Visualisierung von Anfrageergebnissen zuständig ist. Beim Starten der Anfrageverarbeitung wird die Anfrage an den Server übertragen, auf dem Server ausgewertet, und die Anfrageergebnisse werden an den Client zurückgeschickt.

Zur Realisierung von Client/Server-Umgebungen steht in Java die mächtige *Enterprise JavaBeans* Umgebung zur Verfügung. Alle Kommunikationsprozesse zwischen Client und Server werden von dieser Umgebung bereitgestellt, die Realisierung der speziellen Umgebung besteht lediglich in der Implementation der Anwendungslogik. Eine Übersicht über die *Enterprise JavaBeans* Technologie ist in Monson-Haefel (2000) nachzulesen.

Ein weiterer Vorteil der Verwendung von *Enterprise JavaBeans* besteht darin, dass neben der Java-Applikation auf einfache Art weitere Client-Typen unterstützt werden können. Sinnvoll scheint ein HTML-Client und insbesondere ein Java-Applet-Client zu sein. Im letzteren Falle kann die komplette grafische Oberfläche in einem herkömmlichen Browser verwendet werden. Damit steht das Werkzeug ohne jegliche Installation jedem Anwender offen, der eine Anbindung an das Internet besitzt. Allerdings müssen hier Fragen der Zugangsbeschränkung berücksichtigt werden, damit nicht jedem Anwender Zugriff auf alle Korpora gestattet wird.

Interessant ist ein Client/Server-Szenario insbesondere für die Distribution der TIGER-Baumbank. Einem registrierten Benutzer kann in einer solchen Architektur neben der Download-Möglichkeit des Korpus auch ein Online-Zugang zur Suche auf dem Korpus angeboten werden. Und dies von jedem herkömmlichen Browser aus. Ein eingeschränkter Gast-Zugang kann interessierten Anwendern zudem einen ersten Eindruck vom Korpus geben.

13.5 Zusammenfassung

An dieser Stelle wäre eine genaue Auflistung der Laufzeiten sinnvoll, die die verfügbaren Anfragewerkzeuge für typische Korpusanfragen miteinander in Beziehung setzt. Doch bedingt durch die Unterschiede zwischen den einzelnen

Ansätzen, die sich insbesondere in den unterstützten Datenmodell widerspiegeln, ist ein objektiver Laufzeit-Vergleich der Systeme nicht möglich. Zudem sind die Möglichkeiten der Effizienzsteigerung für TIGERSearch noch nicht ausgeschöpft, was eine realistische Bewertung von Laufzeiten erschwert.

Die Daten, die für das VIQTORYA-System zur Verfügung stehen, lassen sich gegenwärtig nicht mit den Laufzeiten von TIGERSearch vergleichen. Erst wenn VIQTORYA auf Korpora mit beliebigen Attributen arbeitet und auch die disjunktive Verknüpfung implementiert ist, wird ein erster Vergleich durchführbar sein. Doch selbst wenn diese Voraussetzungen erfüllt sind, werden die Laufzeiten noch immer nicht vollständig vergleichbar sein, da sie im Falle von VIQTORYA zu stark von der Konfiguration der zugrunde liegenden *mysql*-Datenbank abhängen.

Ein Vergleich mit dem ICECUP-System gestaltet sich ebenfalls schwierig. Da ICECUP auf Baumstrukturen zugeschnitten ist und die Mächtigkeit der grafischen Anfragesprache zudem eingeschränkt ist, ist die Sicherstellung einer effizienten Verarbeitung viel leichter zu bewerkstelligen als im Falle von TIGERSearch. Erst bei der Verarbeitung komplexer Anfragen kann sich zeigen, wo die Stärken und Schwächen der Systeme liegen.

Ein ungefährer Vergleich mit dem Systemen CorpusSearch und tgrep ist hingegen möglich. Auch wenn beide Systeme lediglich auf Baumstrukturen arbeiten, so ist doch die Ausdruckskraft der Anfragesprachen in etwa vergleichbar. Zu diesem Zweck wurden Anfragen auf dem Teilkorpus *Wall Street Journal* der PennTreebank Version 3 auf allen drei Systemen unter den Betriebssystemen Solaris, Linux und für CorpusSearch und TIGERSearch auch unter Windows gestartet. Diese Anfragen reichten von Ein-Knoten-Anfragen bis hin zur Beschreibung komplexerer Strukturen.

Dabei stellte sich zunächst heraus, dass TIGERSearch bis auf einen Randfall schneller arbeitet als CorpusSearch. Dieses Ergebnis ist allerdings wenig überraschend, da CorpusSearch das Korpus für jede Anfrage neu einliest. Der Geschwindigkeits-Unterschied war in denjenigen Fällen besonders gravierend, in denen der Knotenfilter erfolgreich eingesetzt werden konnte.

Im Vergleich zu tgrep ist TIGERSearch in den meisten Fällen deutlich langsamer. Allerdings ist der gemessene Unterschied abhängig vom verwendeten Betriebssystem. So ist die Verarbeitung von TIGERSearch auf der selben Hardware unter Windows um bis zu 50% schneller als unter Linux. Da aber tgrep nicht für die Windows-Plattform verfügbar ist, war ein direkter Vergleich nicht möglich. Bemerkenswert ist, dass TIGERSearch in genau denjenigen Fällen die Verarbeitungsgeschwindigkeit von tgrep erreicht und teilweise sogar übertrifft, in denen der Knotenfilter erfolgreich ist. Dieses Ergebnis erlaubt die begründete Hoffnung, dass der Einsatz fortgeschrittener Filteransätze die Lücke zu tgrep schließen wird.

Insgesamt ist das Laufzeitverhalten von TIGERSearch zufriedenstellend. Insbesondere auf der Windows-Plattform werden durchschnittliche Anfragen in

akzeptabler Zeit verarbeitet. Die Möglichkeiten, eine Anfrageverarbeitung jederzeit abbrechen zu können oder den Suchraum auf einen Teil des Korpus zu beschränken, machen das Werkzeug bei selbst bei langen Auswertungszeiten benutzerfreundlich.

Die Verarbeitung von Anfragen ist damit vollständig beschrieben. Während der Basisalgorithmus eine konsequente Umsetzung des Kalküls darstellt, sorgen die in diesem Kapitel beschriebenen Methoden für eine möglichst effiziente Verarbeitung. Damit ist die Architektur des Suchwerkzeugs bzw. die Gesamtarchitektur des TIGERSearch-Systems vollständig beschrieben. Sie wird in Abb. 13.5 visualisiert.

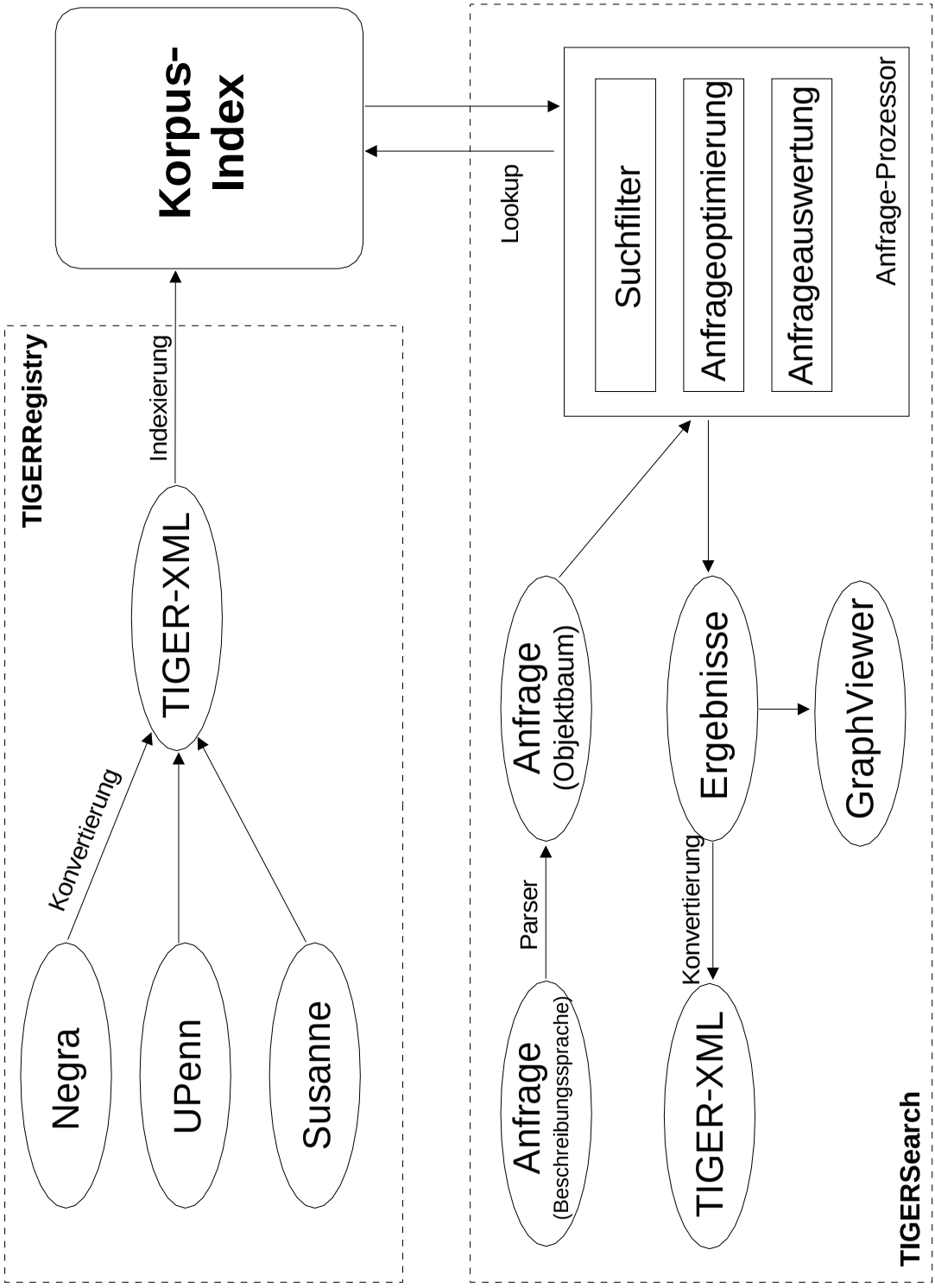


Abbildung 13.5: Gesamtarchitektur des Suchwerkzeugs (Negra, UPenn usw. sind beispielhaft ausgewählte Korpusformate)

Teil IV

Aspekte der Benutzeroberfläche

Kapitel 14

Gestaltung der Benutzeroberfläche

Wie der erste Teil dieser Arbeit gezeigt hat, können eine grafische Benutzeroberfläche im Allgemeinen sowie grafische Unterstützungen bei der Eingabe von Anfragen und der Visualisierung von Ergebnissen im Speziellen die Bedienung eines Anfragewerkzeugs enorm vereinfachen. Beide Hauptbestandteile der TIGERSearch-Architektur – Korpusadministration und Korpussuche – sind deshalb mit einer grafischen Benutzeroberfläche ausgestattet worden. Dieses Kapitel beschreibt ausgewählte Aspekte der TIGERSearch-Benutzeroberfläche, die die Bedienbarkeit in besonderer Weise auszeichnen.

Der erste Abschnitt 14.1 stellt Arbeiten vor, die sich mit der Visualisierung von Baumbanken bzw. mit Eingabehilfen für Baumbank-Anfragesysteme befassen und für die Entwicklung der TIGERSearch-Oberfläche relevant sind. Die Beschreibung ausgewählter Aspekte der TIGERSearch-Benutzeroberfläche beginnt mit der Vorstellung der grafischen Korpusadministration in Abschnitt 14.2. Der nachfolgende Abschnitt 14.3 beschreibt die Integrierte Entwicklungsumgebung zur Verarbeitung von Korpusanfragen. Die dabei verwendeten Strategien zur Visualisierung von Suchanfragen werden in Abschnitt 14.4 vorgestellt. Abschnitt 14.5 beschreibt schließlich ein Konzept zur grafischen Eingabe von Benutzeranfragen.

14.1 Relevante Arbeiten

Obwohl mittlerweile einige Baumbank-Anfragewerkzeuge zur Verfügung stehen (vgl. Kapitel 3), haben Aspekte der Benutzeroberfläche bislang eine untergeordnete Rolle gespielt. Ausnahmen sind die Werkzeuge ICECUP, VIQTORYA und NetGraph, die in Kapitel 3 vorgestellt wurden. Daneben gibt es einige Software-Projekte, die aus anderen Anwendungsfeldern des Baumbank-Bereichs stammen und interessante Konzepte zur Visualisierung von Baumstrukturen verwenden.

ICECUP

Das in Abschnitt 3.4 vorgestellte ICECUP ist ein rein grafisch arbeitendes Suchwerkzeug. Es ist mit einer professionellen Benutzeroberfläche ausgestattet, die ein angenehmes Arbeiten erlaubt. Anfragen werden visuell durch das Zeichnen von Baumstrukturen eingegeben, Anfrage-Ergebnisse werden grafisch als Baumstrukturen angezeigt. Besonders hervorzuheben ist die Möglichkeit, einen Anfrageteilbaum aus einer Baumdarstellung mit der Maus auszuschneiden. Damit kann der Anwender interessante Phänomene auf direktem Wege näher untersuchen.

VIQTORYA

Das VIQTORYA-System ist bereits in Abschnitt 3.5 vorgestellt worden. Es verfügt über eine grafische Oberfläche, mit deren Hilfe Korpusanfragen über eine grafische Eingabehilfe formuliert werden können. Mit Hilfe der Maus können Anfragebausteine aufgebaut und miteinander verknüpft werden. D.h., die logische Anfrage wird in einer vereinfachten Form dargestellt, deren Struktur für den Anwender leichter zu erfassen ist. Eine echte grafische Eingabe von Anfragen wie im Falle von ICECUP kann dieser Zugang jedoch nicht ersetzen.

NetGraph

Das bereits in Abschnitt 3.6 vorgestellte Suchwerkzeug NetGraph zeichnet sich insbesondere durch eine Eingabehilfe zur Formulierung von Suchanfragen sowie eine grafische Visualisierung der aktuell formulierten Suchanfrage aus. Bei der Visualisierung der Anfrageergebnisse wird zudem eine äußerst flexible und leistungsfähige Konzeption verfolgt. Auch im Falle von NetGraph wäre eine echte grafische Eingabe von Suchanfragen eine ideale Abrundung des Werkzeugs.

Annotate

Wie der Name bereits andeutet, handelt es sich bei *Annotate* um ein Annotationswerkzeug (vgl. Brants und Plaehn 2000). Dieses Werkzeug ist zur Annotation von Baumbanken geeignet, die auf dem Negra-Format basieren (vgl. Abschnitt 2.1) und unterstützt damit das in dieser Arbeit verwendete Datenmodell. Annotate wird auch im TIGER-Projekt zur Baumbank-Annotation eingesetzt.

Neben der Annotation kann Annotate auch zur Visualisierung einer Baumbank bzw. über eine Schnittstelle auch zur Visualisierung eines Ausschnitts einer Baumbank und damit zur Präsentation von Anfrageergebnissen eingesetzt

werden (vgl. Abb. 14.1). Das VIQTORYA-System macht von dieser Möglichkeit Gebrauch. Eine gezielte Auszeichnung der matchenden Strukturen ist allerdings nicht möglich. Hervorzuheben ist der Postscript-Export des Werkzeugs, der die Graphvisualisierung in eine Postscript-Datei verwandelt[1].

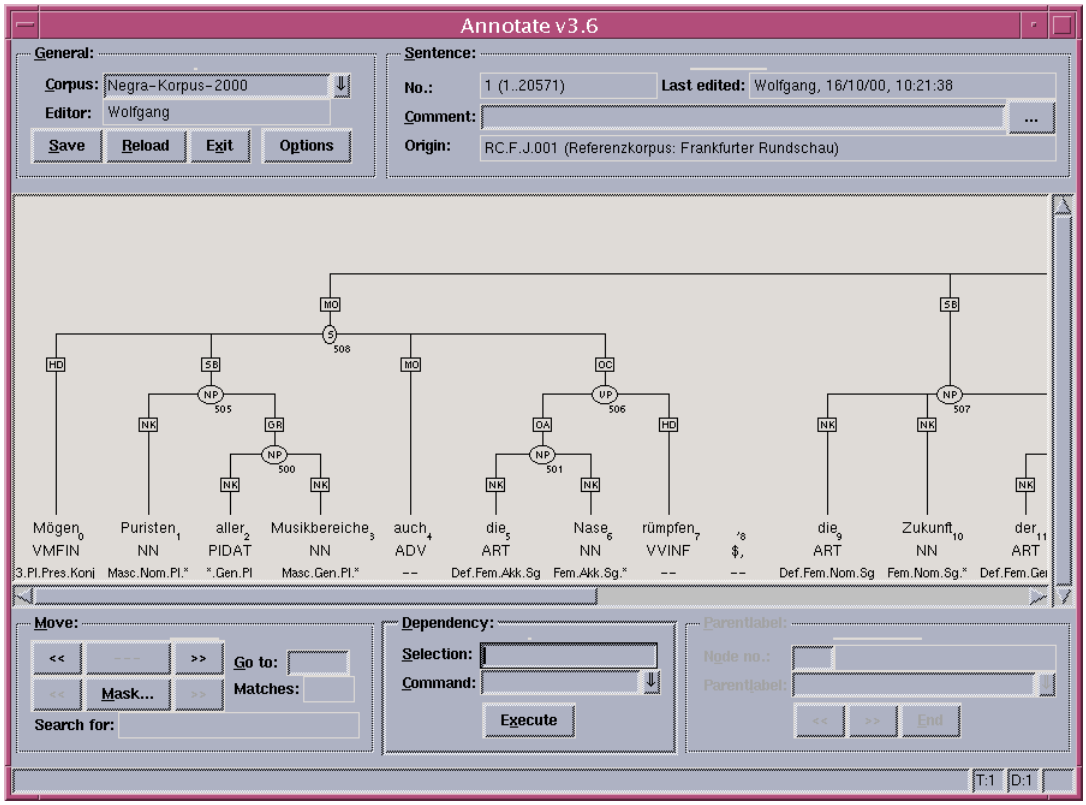


Abbildung 14.1: Darstellung eines Korpusgraphen in Annotate

Charakteristisch für die Visualisierung der Graphstrukturen in Annotate sind die Bottom-Up-Darstellung und die Verwendung von gestuften Kanten (vgl. Abb. 14.1). Sie ergeben sich aus den Eigenheiten des Datenmodells. Da lediglich die Terminale angeordnet sind, wird ihre Reihenfolge durch ihre lineare Abfolge explizit sichtbar. Die Kantendarstellung trägt den kreuzenden Kanten Rechnung, die das Erscheinungsbild bei direktem Durchziehen der Kanten zu stark beeinträchtigen würden. Der Zeichenalgorithmus von Annotate wird in Plaehn (2000) ausführlich beschrieben.

Zur Visualisierung von Anfrageergebnissen in TIGERSearch ist Annotate nur bedingt geeignet. Zum einen fehlt die Hervorhebung der matchenden Knoten, zum anderen ist die Anbindung technisch schwierig, da Annotate in Perl/Tk

[1]Es sei an dieser Stelle angemerkt, dass alle in dieser Arbeit zu findenden Beispielgraphen mit Hilfe des Postscript-Exports von Annotate erzeugt worden sind.

entwickelt wurde und auf einer relationalen Datenbank aufsetzt. Für eine nahtlose Integration ist daher eine Reimplementation der Annotate-Zeichenroutine in Java sinnvoller.

TreeStyle

Bei der Software *TreeStyle* handelt es sich um ein Baumbank-Visualisierungswerkzeug, das auf dem Klammerstrukturformat arbeitet (vgl. Brew 1999). Korpusbäume können hier der Reihe nach eingesehen werden. Die Besonderheit von TreeStyle besteht in der Visualisierung von Teilstrukturen eines Baumes. Dazu kann die Zeichensatz- und Farbeinstellung für jeden Knoten in Abhängigkeit von seiner Annotation verändert werden. Damit lassen sich beispielsweise Köpfe besonders hervorheben. Dieses interessante Konzept ist vergleichbar mit der Formatierung eines XML-Dokuments durch XSLT-Stylesheets. TreeStyle ist als Lisp-Implementation für die Apple Macintosh Plattform verfügbar.

Thistle und Interabora

Thistle ist ein in Java implementiertes System zur Erstellung und Manipulation von linguistischen Diagrammen (Calder 2000b). Ein zur Verfügung stehender Diagrammtyp ist die Darstellung von annotierten Baumstrukturen. Eine Verallgemeinerung auf Graphstrukturen ist zwar geplant, aber bislang nicht realisiert. Eine interessante Möglichkeit der Verarbeitung besteht in der nachträglichen Manipulation der Baumdarstellung. Knoten können beliebig verändert, eingefügt und gelöscht werden. Mit Hilfe der Export-Funktion kann eine Postscript-Version der grafischen Anzeige erzeugt werden. Zur Verwendung des Thistle-Pakets ist nicht notwendigerweise die Installation der Software erforderlich. Alternativ können ausgewählte Diagrammtypen auch über den Web Service *Interarbora* erzeugt und abgerufen werden (vgl. Calder 2000a).

14.2 Korpusadministration (*TIGERRegistry*)

Nach der Vorstellung relevanter Arbeiten werden nun ausgewählte Aspekte der TIGERSearch-Benutzeroberfläche diskutiert. Eine Besonderheit der TIGERSearch-Software besteht darin, dass neben der Korpusanfrage auch die Korpusadministration über eine grafische Benutzeroberfläche erfolgt. Um die Korpora überschaubar zu halten, werden sie hierarchisch organisiert. Die Daten eines Korpus werden in einem Verzeichnis verwaltet, beliebig viele Korpora können unter einem gemeinsamen Verzeichnis gruppiert werden.

Für die Administration der Korpora ist mit der Anwendung *TIGERRegistry* ein eigenständiges Werkzeug entwickelt worden. Durch die Trennung von Korpusadministration und Korpussuche soll die Komplexität der Administration vor denjenigen Anwendern verborgen werden, die ausschließlich von der Korpussuche Gebrauch machen.

Die physikalische Verwaltung der Korpora in Verzeichnissen des Dateisystems spiegelt sich auf der grafischen Oberfläche in Form eines Korpusbaums wider[2] (vgl. Abb. 14.2). Wird in dieser Baumdarstellung ein Korpus angewählt, erscheint eine HTML-Dokumentation, die aus den Metainformationen sowie der Attributdeklaration des Korpus (vgl. Abschnitt 5.2) generiert wird.

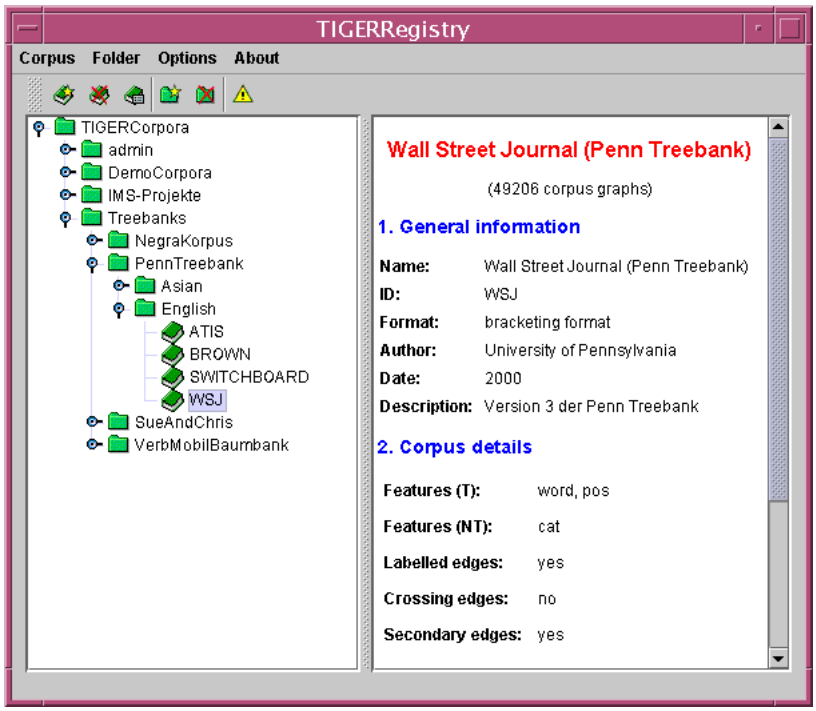


Abbildung 14.2: Hierarchische Organisation der Korpora

Das Einfügen eines neuen Korpus in die Korpussammlung wird ebenfalls grafisch unterstützt. Dazu markiert der Benutzer in einem ersten Schritt das Verzeichnis, in dem das neue Korpus angelegt werden soll. Anschließend spezifiziert er den Korpusaufbereitungs-Prozess in einem separaten Fenster (vgl. Abb. 14.3). Liegt das Korpus wie im abgebildeten Beispiel nicht im TIGER-XML-Format, sondern in einem der unterstützten Baumbank-Formate vor, ist der Aufbereitungsvorgang zweischrittig. In einem ersten Schritt konvertiert ein Importfilter das Originalkorpus in das TIGER-XML-Format, in einem zweiten

[2]Die nachfolgenden Bildschirmabzüge stammen von einer Vorabversion von TIGERSearch 1.1 vom November 2002.

Schritt wird das generierte Korpus indexiert. Das Einstellungsfenster erlaubt die Festlegung der Ausgangsdatei, des zu verwendenden Importfilters und der generierten (temporären) Datei.

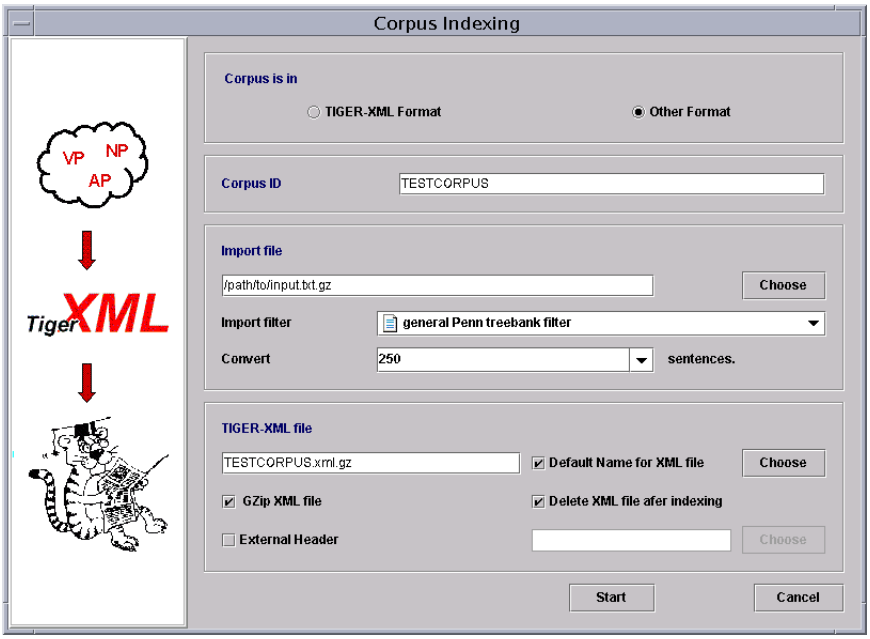


Abbildung 14.3: Einfügen eines neuen Korpus

So angenehm eine grafische Korpusadministration auch ist, eine Verwaltung ausschließlich über die Benutzeroberfläche hat auch Nachteile. Für ein Baumbankprojekt ist es beispielsweise sinnvoll, mindestens einmal am Tag eine automatische Neuaufbereitung der aktuellen Korpusversion vorzunehmen. Dies ist aber nicht möglich, wenn die Aktualisierung nur über die Oberfläche erfolgen kann. Aus diesem Grunde wird bereits an einer Skriptsprache für das TIGERSearch-Werkzeug gearbeitet. Mit Hilfe eines Interpreters können dann Basisaktionen wie das Einfügen und Löschen eines Korpus per Skript durchgeführt werden.

Das folgende Beispiel einer Korpusaktualisierung zeigt, wie die Skriptsprache aussehen wird. In diesem Beispiel wird das zu aktualisierende Korpus zunächst gelöscht und anschließend neu indexiert. Zum Import wird ein Filter für das PennTreebank-Format verwendet.

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>

<ts:tigerscript>

  <ts:delete-corpus corpusid="TEST" />
```

```
  <ts:index-corpus path="/corpora/PennTreebank" corpusid="TEST">
     <ts:input path="/sources/wsj.txt" filter="UPennFilter" />
  </ts:index>

</ts:tigerscript>
```

14.3 Korpussuche (*TIGERSearch*)

Für die Suche auf dem Korpus ist eine eigene Anwendung entwickelt worden, die als *TIGERSearch* bezeichnet wird. Die Idee der Benutzeroberfläche besteht darin, möglichst alle Informationen über verfügbare Korpora, das geladene Korpus usw. in einem gemeinsamen Hauptfenster zugreifbar zu machen. Dieses Konzept der Integrierten Entwicklungsumgebung wird beispielsweise in Programmierumgebungen verwendet.

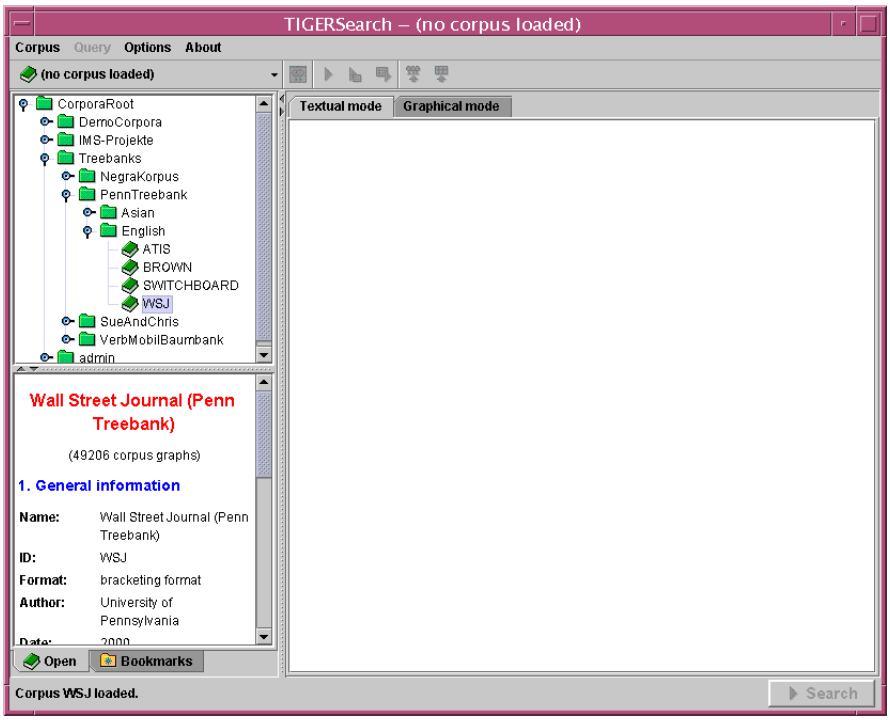


Abbildung 14.4: Das Hauptfenster der TIGERSearch-Anwendung

Das Hauptfenster der Anwendung präsentiert sich nach dem Programmstart zweigeteilt (vgl. Abb. 14.4). Der rechte Bereich ist der so genannte Arbeitsbereich. Hier werden Korpusanfragen formuliert und abgeschickt. Der linke Bereich umfasst die so genannte Informationsleiste. Sie hat die Aufgabe, alle

derzeit über Korpora und das geladene Korpus verfügbare Informationen anzuzeigen. Nach dem Programmstart muss zunächst ein Korpus geladen werden. Zu jedem Korpus wird ein kleiner Hilfetext angegeben, der über den Korpusnamen, die Korpusgröße usw. Auskunft gibt.

Nach dem Laden des Korpus erweitert sich die Informationsleiste um eine weitere Lasche, in der alle Korpusinformationen dargestellt werden (vgl. Abb. 14.5). Hier sind die annotierten Attribute oder Typdefinitionen des Korpus aufgelistet. Durch einen Klick auf ein Attribut wird im unteren Bereich der Leiste eine Liste aller Attributwerte angezeigt. Falls in der Korpusdeklaration angegeben, sind hier auch Erklärungstexte für die einzelnen Attributwerte angegeben. Durch einen Doppelklick auf einen Attributwert wird das entsprechende Attribut-Wert-Paar in den Arbeitsbereich kopiert (z.B. `pos="NN"`).

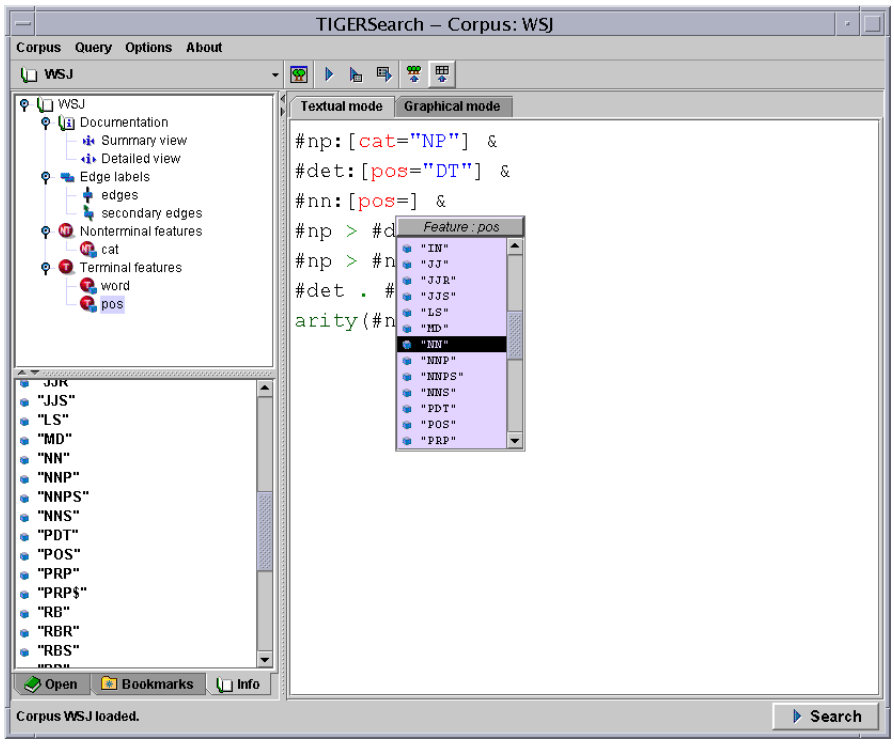


Abbildung 14.5: Das Hauptfenster nach dem Laden eines Korpus

Ein häufiges Problem besteht für den Anwender darin, dass er mit einem Korpus (d.h. mit den annotierten Attributen und dem Annotationsschema) nicht genügend vertraut ist, um sinnvolle Anfragen zu stellen. Hier sollte der Benutzer ohne vorherige Anfrage durch das Korpus blättern können. Dazu wird das zur Visualisierung von Anfrageergebnissen entwickelte Werkzeug *TIGER-GraphViewer* verwendet (vgl. nachfolgenden Abschnitt 14.4).

Die textuelle Eingabe von Anfragen im Arbeitsbereich des Hauptfensters verfügt unter anderem über eine grafische Einfärbung der Sprachsyntax (vgl. Abb. 14.5). Damit bleibt die Darstellung selbst bei komplexen Anfragen übersichtlich. Sehr hilfreich ist auch eine Eingabehilfe, die nach der Eingabe eines Attributnamens mit anschließendem Gleichheitszeichen (z.B. `pos =`) ein PopUp-Fenster anzeigt, in dem alle Attributwerte aufgelistet werden (vgl. Abb. 14.5). Durch Auswahl eines Attributwertes wird dieser in die Anfrage übernommen.

Um die im Laufe der Zeit angesammelten Anfragen archivieren zu können, ist ein Bookmark-Konzept realisiert worden, das sich am Bookmark-Konzept der Internet-Browser orientiert (vgl. Abb. 14.6). Anfragen können hierarchisch in Ordnern organisiert werden. Neben der textuellen Anfrage kann ein kurzer Kommentar eingegeben werden. Auch die Archivierung der Anfrageergebnisse ist möglich. Wie in einem Internet-Browser üblich wird eine Anfrage durch einen Doppelklick auf ihr entsprechendes Symbol aktiviert.

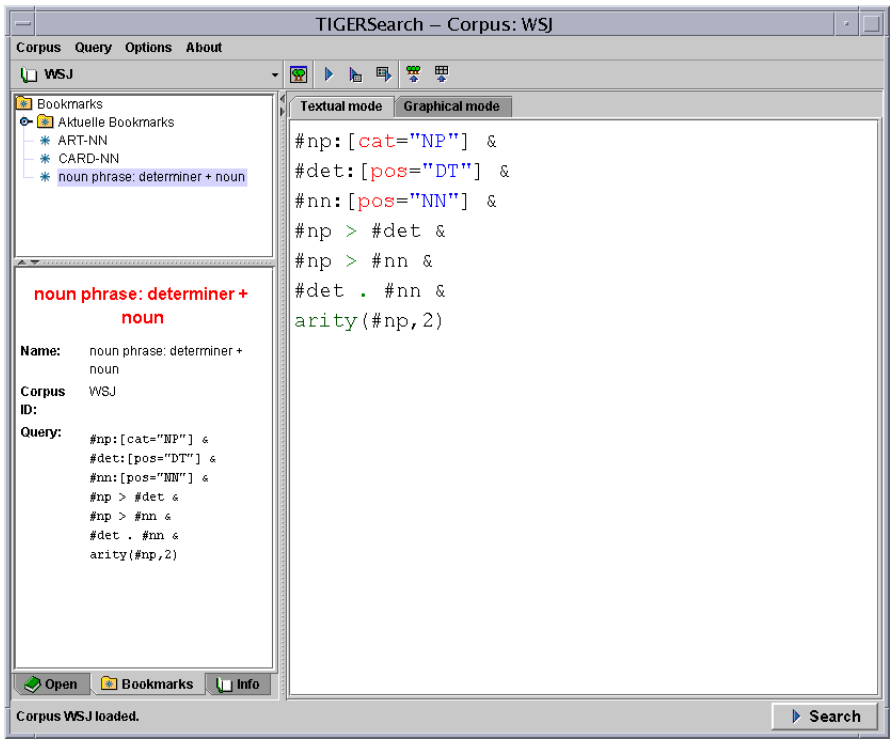


Abbildung 14.6: Ein Bookmark-Konzept für Korpusanfragen

Nach der Eingabe einer Anfrage bzw. dem Abrufen einer als Bookmark vorliegenden Anfrage kann die Anfrageauswertung gestartet werden. Da die Auswertung graphenweise verläuft, wird nach jedem Korpusgraphen ein definierter Zustand erreicht. Deshalb ist ein Abbrechen der Korpusauswertung

durch den Anwender jederzeit möglich. Die Anfrageergebnisse werden im so genannten *TIGERGraphViewer* visualisiert, der im nachfolgenden Abschnitt vorgestellt wird. Hier kann der Anwender durch die matchenden Korpusgraphen blättern. Dabei werden die am Match beteiligten Knoten farblich hervorgehoben. Abb. 14.9 zeigt beispielsweise einen Match der Anfrage aus Abb. 14.5 bzw. Abb. 14.6.

Da das Blättern durch das Korpus nur einen ersten Eindruck von den Matches liefert, steht der so genannte Statistikexport zur Verfügung (vgl. Abb. 14.7). Hier können entsprechende Anfrageknoten und ihre jeweiligen Attributwerte ausgewählt und angezeigt werden. Abb. 14.7 zeigt beispielsweise (nach Frequenz sortiert) alle Nomina, die die Anfrage aus Abb. 14.5 bzw. Abb. 14.6 erfüllen.

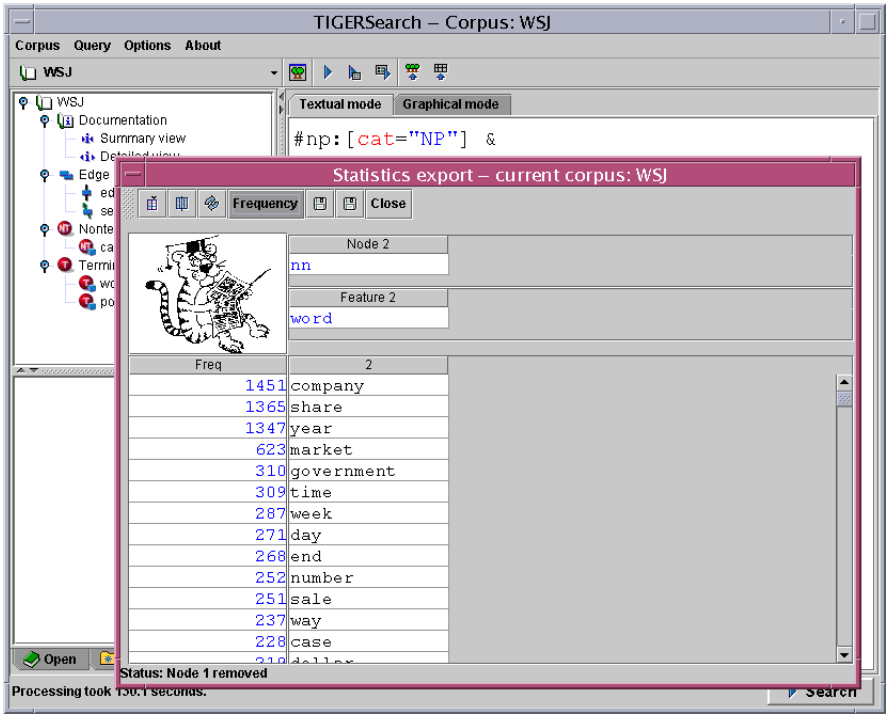


Abbildung 14.7: Der Statistik-Export

Abschließend sei noch ausgeführt, dass die Benutzeroberfläche mittlerweile vollständig internationalisiert ist. Beliebige Unicode-Zeichen können bei der textuellen Anfrage eingegeben, bei der Anfrageauswertung berücksichtigt und bei der Visualisierung der Anfrageergebnisse angezeigt werden. Abb. 14.8 zeigt eine Suchanfrage und ein entsprechendes Suchergebnis auf der *Penn Chinese Treebank* (vgl. Xue et al. 2002).

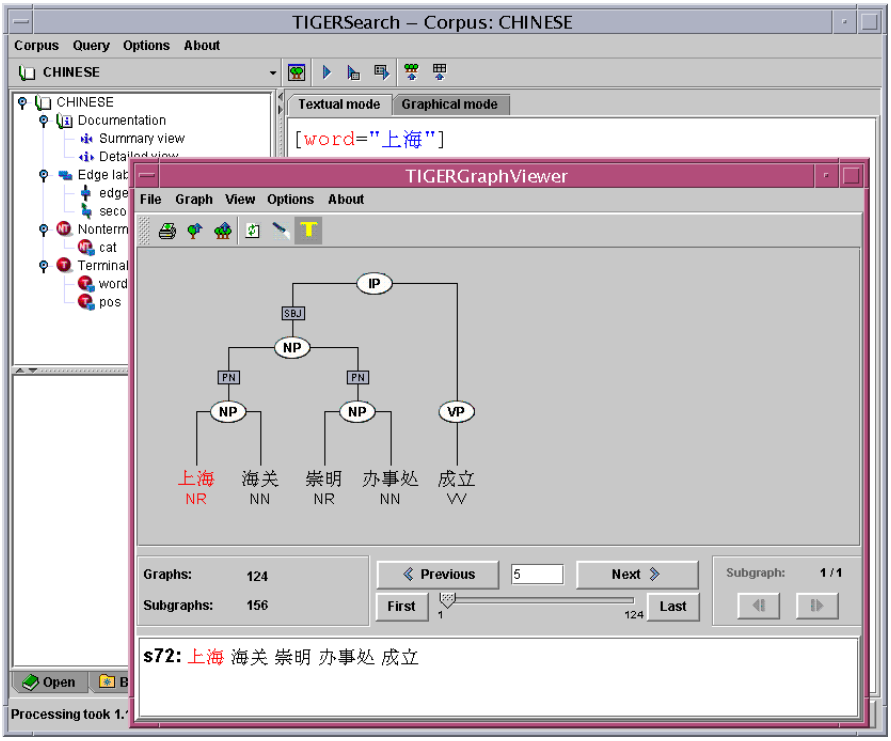


Abbildung 14.8: Internationalisierung der Benutzeroberfläche

Es bleibt festzuhalten, dass die Benutzeroberfläche zur Korpussuche mittlerweile auch professionellen Ansprüchen gerecht wird. In Kombination mit der Visualisierung von Anfrageergebnissen ist hier ein sehr intuitives und angenehmes Arbeiten möglich.

14.4 Ergebnisvisualisierung (*TIGERGraphViewer*)

Ein Viewer für Anfrageergebnisse

Ein Korpusanfragewerkzeug ist nur mit einer geeigneten Visualisierung der Anfrageergebnisse wirklich benutzerfreundlich. Wichtig ist dabei eine möglichst nahtlose Integration. Wie das Beispiel VIQTORYA zeigt, ist die Verwendung einer externen Applikation zwar ein möglicher Weg, die Anzeige der Ergebnisse ist aber immer sehr umständlich. Da die verfügbaren Baumbankvisualisierungs-Werkzeuge nicht auf Graphstrukturen arbeiten, musste für die TIGERSearch-Software eine eigene Komponente entwickelt werden. Glücklicherweise stand aber eine Beschreibung der Graphzeichenroutine von Annotate zur Verfügung (Plaehn 2000), so dass der Zeichenalgorithmus nicht neu entwickelt, sondern lediglich reimplementiert werden musste.

Das entstandene Werkzeug wird *TIGERGraphViewer* genannt. Es wird zur Visualisierung von Anfrageergebnissen (vgl. Abb. 14.9), aber auch zur Exploration des Korpus ohne vorherige Korpusanfrage eingesetzt (vgl. Abschnitt 14.3). Neben der Navigation durch das Korpus bzw. durch die matchenden Korpusgraphen stehen das Implodieren von Subgraphen (d.h. das Zusammenziehen eines Subgraphen zu einem stellvertretenden Knoten) sowie die Fokussierung auf den matchenden Subgraphen zur Verfügung. Eine sinnvolle Erweiterung wäre ein Stylesheet-Konzept wie im TreeStyle-Werkzeug, das eine Anpassung des Erscheinungsbildes eines Graphen in Abhängigkeit von seiner Annotation erlaubt.

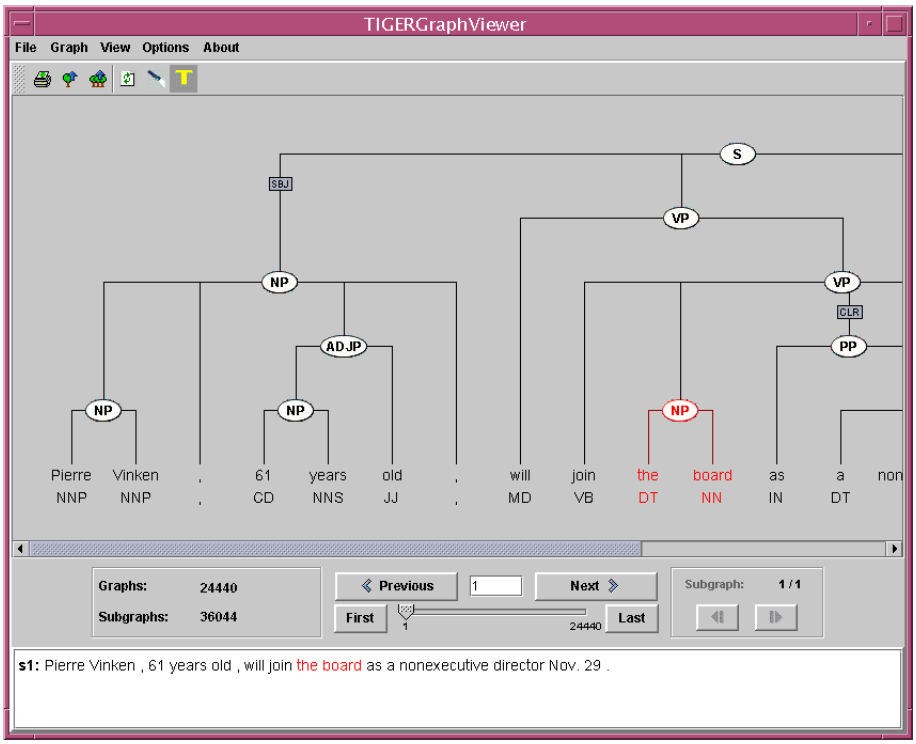


Abbildung 14.9: Visualisierung der Anfrageergebnisse im TIGERGraphViewer

Das Grafikformat SVG

Der Export der Graphdarstellung zur Verwendung in anderen Programmen basiert auf dem Grafikformat *SVG*. Das *Scalable Vector Graphics* Format SVG ist ein XML-basiertes Format zur Repräsentation zweidimensionaler Grafiken (vgl. Ferraiolo 2001). Die zentrale Idee besteht darin, eine Grafik aus skalierbaren Grafikprimitiven zusammenzusetzen, die durch XML-Elemente beschrieben werden. Durch die Verwendung von XML als Basisformat sind SVG-Grafiken mit

den bekannten XML-Techniken generierbar, editierbar und manipulierbar. Mittlerweile wird SVG auch von kommerziellen Produkten (z.B. Corel Draw oder Adobe Illustrator) unterstützt. Das folgende XML-Dokument stellt eine vollständige SVG-Grafik dar, die ein an den Ecken abgerundetes Rechteck enthält:

```
<?xml version="1.0" encoding="UTF-8"?>
<svg>
  <rect x="50" y="50" rx="5" ry="5" width="150" height="50" />
</svg>
```

Durch die Verwendung von SVG als Exportformat ergeben sich zahlreiche konzeptuelle Vorteile:

- SVG-Grafiken sind XML-Dokumente und können als solche mit jedem Texteditor nachträglich bearbeitet werden. Stellt der Anwender in seiner Grafik einen Fehler in der Annotation fest, so muss er nicht das Korpus neu aufbereiten und seine Anfrage neu stellen, sondern lediglich die SVG-Grafik entsprechend abändern.
- Möchte der Anwender nicht den gesamten Korpusgraphen darstellen, sondern lediglich einen Ausschnitt, so kann er die nicht benötigten Subgraphen aus der SVG-Datei entfernen.
- Die Darstellung einer SVG-Grafik ist durch Cascading Stylesheets (CSS) oder XSLT-Stylesheets sehr leicht an die eigenen Bedürfnisse anpassbar. So kann beispielsweise das Match-Highlighting durch ein einfaches Cascading Stylesheet ausgeschaltet werden.
- Durch die Skalierbarkeit lassen sich die SVG-Grafiken in einem SVG-Viewer beliebig vergrößern und verkleinern. Die grafische Darstellung ist im Gegensatz zu einer Rastergrafik ohne Qualitätsverlust möglich.
- Falls erforderlich, kann die SVG-Grafik in andere Grafikformate wie JPG, TIFF oder Dokumentformate wie PDF verwandelt werden.

Export eines Korpusgraphen als SVG-Grafik

Um die Verarbeitung der Grafik durch Stylesheets oder andere Programme zu erleichtern, wird die SVG-Darstellung eines Korpusgraphen in Gruppen eingeteilt, die in SVG durch ein eigenes Gruppierungs-Element `<g>` repräsentiert werden. Mit zusätzlichen Kommentaren bleibt die Grafik auch für die manuelle Sichtung lesbar. Das folgende Beispiel zeigt eine SVG-Grafik für einen Syntaxgraphen, der aus der Nominalphrase *Pierre Vinken* besteht (vgl. Korpusgraph in Abb. 14.9). Damit die Darstellung übersichtlich bleibt, fehlen an dieser Stelle alle Attribute zur Zeichensatz- und Farbformatierung.

```
<?xml version="1.0" encoding="UTF-8"?>

<svg>
    <!-- "s1_1":[word="Pierre" & pos="NNP"] -->
    <g type="t" id="s1_1" match="subgraph">
      <text x="55" y="336" text-anchor="middle">Pierre</text>
      <text x="55" y="357" text-anchor="middle">NNP</text>
    </g>

    <!-- "s1_2":[word="Vinken" & pos="NNP"] -->
    <g type="t" id="s1_2" match="subgraph">
      <text x="115" y="336" text-anchor="middle">Vinken</text>
      <text x="115" y="357" text-anchor="middle">NNP</text>
    </g>

    <!-- "s1_150":[cat="NP"] -->
    <g type="nt" id="s1_150" match="node">
      <!-- Kante "s1_150" > "s1_1" -->
      <g type="edge">
        <line x1="55" y1="321" x2="55" y2="271" />
      </g>
      <!-- Kante "s1_150" > "s1_2" -->
      <g type="edge">
        <line x1="115" y1="321" x2="115" y2="271" />
      </g>
      <g>
        <line x1="55" y1="271" x2="115" y2="271" />
        <ellipse cx="85" cy="271" rx="16" ry="10" />
        <text x="85" y="275" text-anchor="middle">NP</text>
      </g>
    </g>
</svg>
```

Um den GraphViewer auch in anderen Projekten einsetzen zu können, wird als Datenschnittstelle das TIGER-XML-Format verwendet (vgl. Unterabschnitt 8.2.2). Damit können ein im TIGER-XML-Format kodiertes Korpus bzw. ausgewählte Korpusgraphen mit Matchinformation vom GraphViewer dargestellt werden.

Interaktive SVG-Grafiken

Eine interessante Eigenschaft von SVG ist die Möglichkeit, die grafische Darstellung zu animieren. Animationen können zeitgesteuert kontrolliert oder durch

Benutzeraktionen ausgelöst werden. Das folgende Beispiel zeigt, wie die Farbe eines Rechtecks durch das Bewegen der Maus auf das Rechteck abgeändert wird. Es unterstreicht, mit welchen einfachen Mitteln Interaktionen aufgebaut werden können.

```
<?xml version="1.0" encoding="UTF-8"?>

<svg>
  <rect x="50" y="50" rx="5" ry="5"
        width="150" height="50"
        style="fill:red; stroke:black;">

    <set begin="mouseover" end="mouseout"
         attributeName="fill" from="red" to="blue" />

  </rect>
</svg>
```

Eine naheliegende Idee besteht darin, aus dem SVG-Export eines einzelnen Korpusgraphen eine graphenweise Darstellung aller matchenden Graphen zusammenzusetzen. Die dazu verwendeten Techniken sind prinzipiell mit dem oben angegebenen Beispiel vergleichbar. Die Navigation durch die Korpusgraphen erfolgt durch eine separate Navigationsleiste. Durch die Möglichkeit, SVG-Dateien auch komprimiert zu erzeugen, bleibt die Dateigröße in akzeptablen Grenzen. Eine interaktive SVG-Grafik kann nicht nur mit einer SVG-Viewer-Applikation betrachtet werden. Mit Hilfe eines SVG-Plugins kann die Grafik auch in einem WWW-Browser dargestellt oder in Präsentationprogramme wie PowerPoint eingebunden werden (vgl. Abb. 14.10).

Der Export interaktiver SVG-Grafiken stellt einen sehr leistungsfähigen Ansatz zur Visualisierung von Anfrageergebnissen dar, der unabhängig von der TIGERSearch-Software ist. Interessante Korpusgraphen oder Anfrageergebnisse lassen sich zwischen den Anwendern über das Internet austauschen und in einem beliebigen WWW-Browser darstellen.

14.5 Grafische Eingabe von Suchanfragen (*TIGERin*)

Für die grafische Eingabe von Suchanfragen hat Holger Voormann im Rahmen einer Diplomarbeit das Werkzeug *TIGERin* zum Zeichnen von Anfragen entwickelt (vgl. Voormann 2002; Voormann und Lezius 2002). In diesem Werkzeug werden Graphbeschreibungen aus den visuellen Elementarbausteinen Knoten und Relationen aufgebaut.

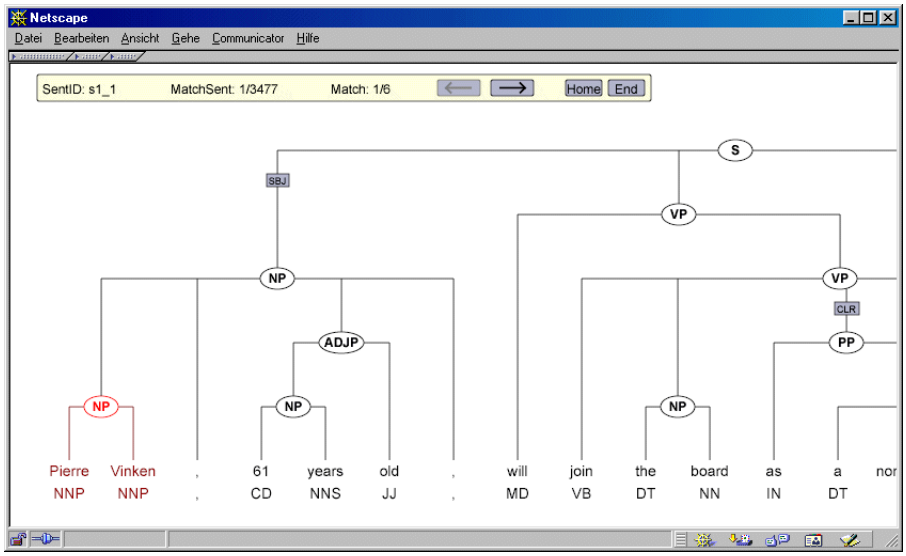


Abbildung 14.10: Darstellung der interaktiven SVG-Grafik in einem Browser

Die Arbeitsfläche des Werkzeugs ist in zwei Bereiche unterteilt (vgl. Abb. 14.11). Im oberen Bereich werden Nichtterminale, im unteren Bereich Terminale (Token) dargestellt. Durch einen Mausklick in den entsprechenden Bereich wird ein entsprechender Knoten erzeugt. Im Inneren eines Knotens wird die Attributspezifikation durch Attribut-Wert-Paare beschrieben. Zur weiteren Spezifikation können Attribut-Wert-Paare durch Konjunktion und Disjunktion miteinander verknüpft werden. Für Attribute, bei denen eine Auflistung aller Attributwerte nicht verfügbar ist, kann eine direkte Eingabe des Attributwerts vorgenommen werden (z.B. bei der Wortform).

Relationen werden durch das Erzeugen von Linien zwischen Knoten erstellt. Eine Dominanzbeziehung beginnt am Halboval an der unteren Kante eines Knotens und endet am Halbkreis an der oberen Kante eines zweiten Knotens. Der Relationstyp lässt sich am Kanten-Pulldownmenü ändern. Eine durchgezogene Linie symbolisiert eine direkte Dominanzbeziehung, eine gestrichelte Linie steht für eine allgemeine Dominanzbeziehung. Ebenfalls über ein Pulldownmenü werden optionale Kantenbeschriftungen bzw. Entfernungsangaben ausgewählt. Linien zwischen den seitlichen Kanten zweier Knoten symbolisieren eine Präzedenzbeziehung. Prädikate werden direkt am Knoten über ein Menü spezifiziert.

Abb. 14.11 zeigt, wie eine Anfrage aus Knoten und Relationen zusammengesetzt werden kann. Die Nominalphrase ist beliebig tief unter dem Satzknoten eingebettet und umfasst drei direkte Nachfolger. Durch die Angabe der Stelligkeit ist die Nominalphrase auf genau drei Nachfolger festgelegt. Die Verbalphrase muss kontinuierlich sein und enthält ein finites Voll- oder Hilfsverb.

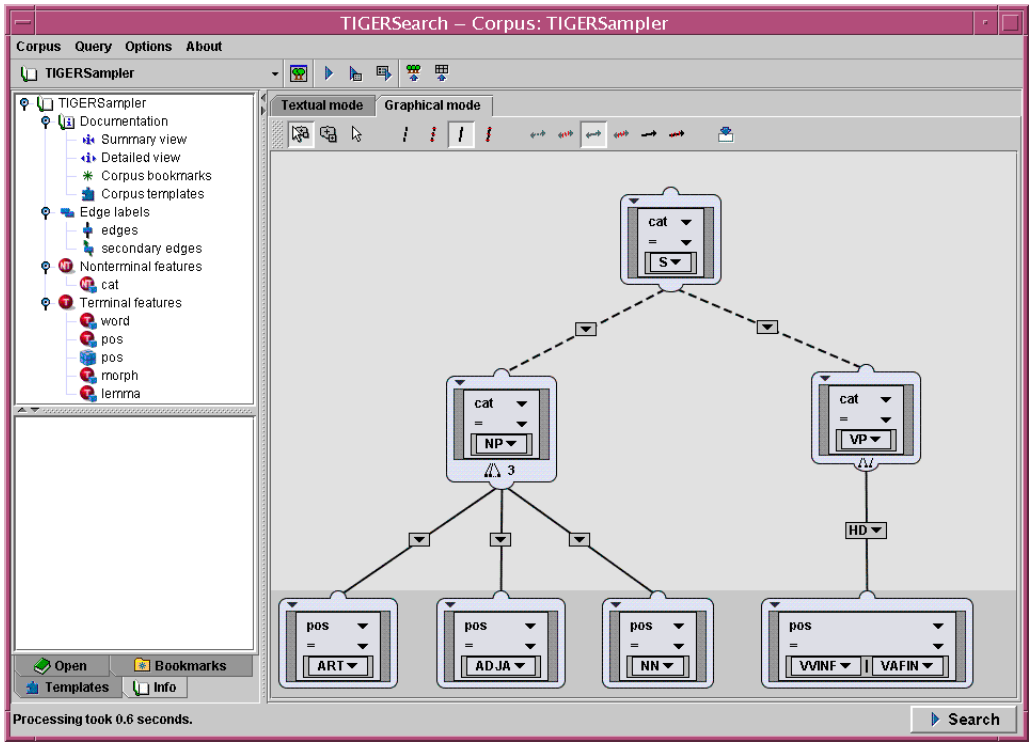


Abbildung 14.11: Grafische Eingabe von Benutzeranfragen

Aus der grafischen Darstellung wird eine textuelle Anfrage generiert, die ggf. weiter bearbeitet oder ausgewertet werden kann (vgl. Abb. 14.11):

```
#n1:[cat="S"] >* #n2:[cat="VP"] &
#n1 >* #n3:[cat="NP"] &
#n2 >HD [pos=("VVINF" | "VAFIN")] &
continuous(#n2) &
#n3 > [pos="ART"] &
#n3 > [pos="ADJA"] &
#n3 > [pos="NN"] &
arity(#n3, 3)
```

Mit Hilfe der visuellen Eingabe von Anfragen können einfache bis durchschnittlich komplexe Anfragen spezifiziert werden. Insbesondere für Gelegenheitsbenutzer stellt dieser Zugang eine enorme Erleichterung dar, da nicht das Erlernen der Anfragesprache, sondern die explorative Arbeit mit dem Korpus bei der Einarbeitung in das Suchwerkzeug im Vordergrund steht.

Das Werkzeug zur grafischen Eingabe von Suchanfragen ist bereits vollständig in die Grafische Oberfläche integriert worden und stellt dort einen zur textuellen Anfrage-Eingabe alternativen Eingabe-Client dar (vgl. Abb. 14.11).

14.6 Zusammenfassung

Zusammenfassend lässt sich festhalten, dass die Gestaltung der Benutzeroberfläche maßgeblich zur Benutzerfreundlichkeit der TIGERSearch-Software beigetragen hat. Die grafische Eingabe von Suchanfragen und die Ansätze zur Korpusvisualisierung geben dem Anwender darüber hinaus Hilfsmittel an die Hand, um explorativ an ein Korpus heranzugehen.

Teil V

Zusammenfassung

Kapitel 15

Zusammenfassung und Diskussion

15.1 Zentrale Ergebnisse

Die vorliegende Arbeit beschreibt die Konzeption und Implementation einer Software zur Suche auf syntaktisch annotierten Korpora. Die zentralen Ergebnisse dieser Arbeit werden in diesem Abschnitt noch einmal herausgestellt. Für eine ausführlichere Zusammenfassung der wesentlichen Beiträge dieser Arbeit sei auf Lezius (2002) verwiesen.

Der erste Teil der Arbeit spezifiziert zunächst die Anforderungen, die an das Suchwerkzeug gestellt werden. Von zentraler Bedeutung ist dabei das zu unterstützende Datenmodell, das sich nach der TIGER-Baumbank richtet, in dessen Kontext diese Arbeit entstanden ist. Das Datenmodell sieht danach schwache Graphstrukturen vor, die sich gegenüber den in anderen Arbeiten verwendeten Baumstrukturen vor allem durch das Zulassen kreuzender Kanten unterscheiden. Dieses Datenmodell kann nur begrenzt in existierenden Baumbank-Kodierungsformaten dargestellt werden. Daneben wird das Datenmodell nur vom Baumbank-Suchwerkzeug VIQTORYA unterstützt, das zur Zeit auf Korpora im Negra-Format eingeschränkt ist. Aus diesen Gründen hat die vorliegende Arbeit zwar von den Ideen anderer Arbeiten profitiert, die in diesen Projekten entwickelten Ergebnisse konnten jedoch nicht übernommen werden.

Im zweiten Teil der Arbeit wird eine Korpusbeschreibungssprache entwickelt, die sowohl zur Korpusdefinition als auch zur Korpusabfrage verwendet wird. Die Sprache genügt den linguistischen Anforderungen und ist mit einer formalen Syntax und Semantik umfassend dokumentiert. Für die Korpusdefinition ist ein semantisch äquivalentes XML-basiertes Kodierungsformat entwickelt worden. Das so genannte TIGER-XML-Format kodiert nicht nur die Graphen eines Korpus, sondern erlaubt zudem die Kodierung der Ergebnisse einer Korpusanfrage. Es stellt damit eine flexible Grundlage zum Import und Export von Korpusgraphen dar. Durch die umfassende XML-Unterstützung der verwendeten Programmiersprache Java konnten leistungsfähige Funktionalitä-

ten wie die Konvertierung in andere Korpusformate durch XSLT-Stylesheets in die Software integriert werden.

Der dritte Teil der Arbeit zeigt die Konzeption und Implementation der Anfrageverarbeitung. Das Konzept wird durch einen Kalkül, d.h. einen syntaktischen Ableitungsbegriff definiert. Zentrale Bausteine des Kalküls sind die Umwandlung der Anfrage in eine Disjunktive Normalform, die Verwendung eines Haldenmodells zur Repräsentation logischer Variablen und die nichtdeterministische Auswahl des Korpusgraphen bzw. des Korpusknotens.

Die Implementation des Kalküls besteht aus zwei Teilen. Der erste Teil umfasst die Repräsentation der Korpusdefinition und daraus direkt ableitbarer Aussagen in Form eines Korpus-Index. Der zweite Teil stellt die eigentliche Anfrageverarbeitung dar, die erst zur Laufzeit durchgeführt werden kann. Die Verarbeitung zur Laufzeit wird durch einen Backtracking-Algorithmus realisiert, der den Nichtdeterminismus des Kalküls durch eine erschöpfende Suche auflöst. So werden alle Korpusgraphen der Reihe nach durchlaufen und für den Knotentest stets alle Graphknoten berücksichtigt. Damit diese aufwändige Suchstrategie nicht zu ineffizienter Anfrageverarbeitung führt, reduzieren Filterstrategien den Suchraum. Optimierungen der Anfrageanordnung bringen die Anfrage in eine Reihenfolge, die der implementierten Suchstrategie möglichst weit entgegen kommt. Einige eher technische Erweiterungen der Implementation unterstützen zusätzlich die Effizienz des Systems.

Den Abschluss der Arbeit stellt die Vorstellung zentraler Aspekte der Benutzeroberfläche dar. Dieser Bereich der Mensch-Maschine-Schnittstelle muss bei der Entwicklung von Forschungs-Prototypen leider oftmals vernachlässigt werden, obwohl er maßgeblich über die Praxistauglichkeit eines Werkzeugs entscheidet. Die grafische Eingabe von Benutzeranfragen und die Korpusvisualisierung in Form einer Viewer-Applikation bzw. einer (interaktiven) SVG-Grafik sind zwei herausragende Punkte, die die Benutzeroberfläche von TIGERSearch in besonderer Weise auszeichnen.

Insgesamt wurde ein praxistaugliches Werkzeug entwickelt, dessen Konzeption mit formalen Methoden abgesichert ist und zugleich über eine grafische Benutzeroberfläche verfügt, die ein intuitives Arbeiten ermöglicht.

15.2 Ansätze zur Weiterentwicklung

Templates

Innerhalb des Suchwerkzeugs stellt die Korpusbeschreibungssprache den ausgereiftesten Bestandteil der Systemarchitektur dar. Hier fehlt zur Zeit noch die Umsetzung der Templates. Zwar sind Templates bereits Bestandteil des formalen Entwurfs der Sprache, doch ist die Integration in die Anfrageauswertung noch nicht vollständig realisiert.

Das anvisierte Konzept sieht vor, Templateaufrufe innerhalb einer Anfrage noch während des Parsens der Anfrage und damit noch vor der eigentlichen Anfrageauswertung aufzulösen und durch den Templaterumpf (bei entsprechenden angepassten Variablenbindungen) zu ersetzen. Da sich Templates nicht selbst aufrufen dürfen (sei es direkt oder indirekt über ein weiteres Template), kann es keine zyklischen Templateaufrufe geben. Der Anfrage-Parser beinhaltet damit einen Template-Compiler, der eine semantisch äquivalente Anfrage ohne enthaltene Templateaufrufe erzeugt.

Allquantor

Nachdem das TIGERSearch-Werkzeug in einigen Projekten bereits recht intensiv genutzt wird, hat sich relativ rasch der Wunsch der Anwenderschaft nach der Integration eines Allquantors ergeben. Ein Allquantor würde die Mächtigkeit der Sprache natürlich deutlich ausbauen. Die dabei auftretenden Probleme einer nahtlosen Integration in die Korpusbeschreibungssprache, einer effizienten Implementation und auch einer geeigneten Visualisierung bei der grafischen Anfrageeingabe müssen jedoch zuvor diskutiert und gelöst werden. Erste Ansätze zur Konzeption eines Allquantors werden bereits diskutiert, die Integration eines Allquantors ist für die nahe Zukunft geplant.

Effiziente Suche auf großen Korpora

Bei der Verbesserung der Effizienz der Anfrageverarbeitung wird der Ausbau des Index einen Schwerpunkt darstellen. Die Integration weiterer Filteransätze, der Einsatz weiterführender Kompressionstechniken zur Reduzierung des Hauptspeicherbedarfs und das blockweise Einlesen eines Index zur Verarbeitung beliebig großer Korpora sind drei zentrale Punkte, deren Konzeption zwar im Rahmen dieser Arbeit bereits angedeutet wurde, deren geschickte Implementation aber umfangreiche Arbeiten nach sich ziehen wird. Erst mit der Realisierung dieser Konzepte wird die Effizienz der Suche auf größeren Korpora ausreichend sein.

XML Query

Die modulare Architektur des Systems, die Implementation in der Programmiersprache Java und die Verwendung aktueller Standards wie XML und XSLT erlauben nicht nur eine weitere Verbesserung der Implementation, sondern auch Erweiterungen in völlig neue Richtungen. Die Entwicklungen in diesen Bereichen befinden sich so stark im Fluss, dass die kommenden Trends nicht vorhersagbar sind.

Insbesondere der Standardisierungsprozess im Bereich der XML-Anfragesprachen wird für die Anfrageauswertung von TIGERSearch von großem Interesse sein. Interessant ist beispielsweise die Frage, ob sich die hier definierte Anfragesprache auf eine XML Query Anfrage abbilden lässt. Auf diese Weise könnte ein alternativer Anfrageprozessor entwickelt werden, der auf der TIGER-XML-Variante des Korpus arbeitet.

Java-basierte Ansätze

Aus technischer Sicht sind Weiterentwicklungen möglich, die die Eigenschaften der verwendeten Programmiersprache Java konsequent ausnutzen. Wie bereits ausgeführt, sind hier Client/Server-Architekturen denkbar, die zu einer Verschmelzung des TIGERSearch-Werkzeugs mit dem TIGER-Korpus führen. Strategien zur effizienten Verarbeitung wie die Parallelisierung der Anfrageverarbeitung finden dabei serverseitig statt. Angesichts der rasanten Entwicklung im Bereich der Web Services könnte ein solches Szenario maßgeblich zur Verbreitung des Korpus und damit auch des Suchwerkzeugs beitragen.

15.3 Diskussion

Die TIGERSearch-Software ist im Januar 2002 der Forschungsgemeinschaft zur Verfügung gestellt worden. In den ersten 12 Monaten seit der Veröffentlichung sind etwa 120 Lizenzen beantragt worden. Erste Rückmeldungen der Benutzer sind durchweg positiv, fundierte Aussagen sind aber nur durch eine Benutzerbefragung zu begründen. Eine solche Befragung ist geplant.

Erste Erfahrungen beim Einsatz der Software liegen aus dem TIGER-Projekt vor, in dem bereits im Sommer 2001 Vorläuferversionen der TIGERSearch-Software verwendet wurden. Eine zentrale Verwendung der Software im TIGER-Projekt ist das phänomenbasierte Retrieval. Hier suchen Korpusannotierer zur Unterstützung bei der Annotation eines Phänomens nach Korpusbelegen, um sich für eine Annotationsvariante zu entscheiden. Neben der Unterstützung der Annotation soll die Korpussuche auch bei der Dokumentation des Korpus im Annotationsschema eine zentrale Rolle spielen (vgl. Albert et al. 2002). Neben der Erläuterung, wie ein Phänomen im Korpus annotiert wird, soll jeweils auch eine Suchanfrage angegeben werden, die Korpusbelege für dieses Phänomen beschreibt. Es ist damit festzuhalten, dass das TIGERSearch-Werkzeug im Rahmen des TIGER-Projekts erfolgreich eingesetzt werden konnte.

Mit der Freigabe des TIGER-Korpus an die Forschungsgemeinschaft wird ein Benutzerkreis zur Verfügung stehen, der die besonderen Eigenschaften der Software (insbesondere das graphenbasierte Datenmodell) ausnutzen wird. Ei-

ne Evaluationsstudie für diesen Nutzerkreis wird Aufschluss über Stärken und Schwächen des Konzepts geben und die weitere Entwicklung bestimmen.

Für eine abschließende Diskussion dieser Arbeit bleibt zu prüfen, ob die zu Beginn festgelegten Anforderungen erfüllt worden sind (vgl. Abschnitt 1.3). Die Unterstützung des graphenbasierten Datenmodells als grundlegende Forderung ist vollständig realisiert worden. Die Ausdruckskraft der Anfragesprache hingegen wird durch einen Allquantor noch abgerundet werden. Für Gelegenheitsanwender stellt die grafische Eingabe eine interessante Alternative dar, da sich der Anwender auf das Korpus konzentrieren kann und das Erlernen der Anfragesprache in den Hintergrund rückt.

Bei der effizienten Verarbeitung großer Korpora besteht noch Handlungsbedarf. Durch die Implementation einiger noch ausstehender Arbeiten am Index wird jedoch die grundsätzliche Verarbeitbarkeit großer Korpora hergestellt werden. Ob die geplanten Filterstrategien auch eine effiziente Verarbeitung erlauben werden, bleibt abzuwarten. Für manuell annotierte Baumbanken, die einen Umfang von wenigen Millionen Token nicht überschreiten, ist jedoch bereits zum gegenwärtigen Zeitpunkt ein effizientes Arbeiten auf einem handelsüblichen PC möglich.

Eine besondere Eigenschaft des TIGERSearch-Werkzeugs ist das Bemühen um eine angenehme Arbeitsumgebung. Die grafische Eingabe von Suchanfragen und die Visualisierung von Anfrageergebnissen schaffen eine integrierte Applikation, die eine explorative Arbeitsweise erlaubt.

Insgesamt ist festzuhalten, dass das implementierte System den Anforderungen, die eine manuell annotierte Baumbank wie das TIGER-Korpus stellt, voll gerecht wird. Die weiteren Arbeiten werden die bestehende Lücke für große Korpora schließen.

Abstract

A search tool for syntactically annotated corpora

1 Introduction

Syntactically annotated corpora, or so-called *treebanks*, have established themselves as an important part of linguistic research. They can form the basis for the development of statistical language models or the verification of theoretical assumptions. Well-known treebanks for English are the *Penn Treebank* (Taylor et al. 2000) and the *Susanne Corpus* (Sampson 1995).

For German, two treebanks are currently available. The *VerbMobil treebank* is a speech corpus that comprises 250,000 tokens (Hinrichs et al. 2000). The treebank consists of annotated syntactic tree structures based on transcribed dialogs in the scenarios of appointment negotiations, travel arrangements, and personal computer maintenance. The *Negra Corpus* comprises 20,000 syntactically annotated newspaper sentences (about 350,000 tokens). Characteristics of the Negra annotation scheme are the labelling of edges to express relations between nodes and their children, and the use of crossing edges to describe long-distance dependencies such as extraposed relative clauses (cf. fig. 1).

The *TIGER* project is based on the resources of the Negra project. The aims of the TIGER project are to refine the Negra annotation scheme and to annotate another 40,000 German newspaper sentences (Brants et al. 2002). For applications, this wealth of information can only be exploited by a specialized search tool. Within the TIGER project, a treebank search engine has been developed which supports the rather complex data model of the TIGER corpus (cf. fig. 1). This tool is called *TIGERSearch*.

This paper describes the concept of the TIGERSearch tool. The first section deals with work related to the development of the TIGERSearch tool. It presents treebank encoding formats and treebank search tools which have influenced the

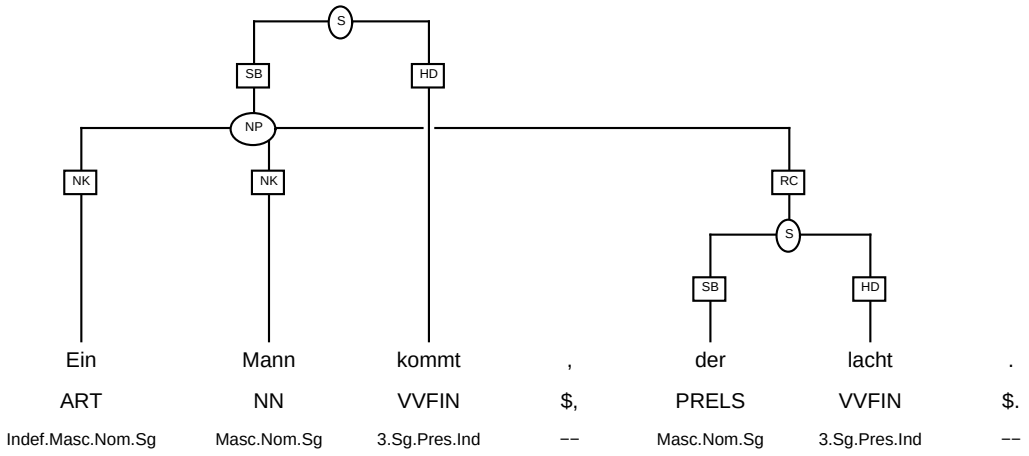


Figure 1: Annotation of an extraposed relative clause

development of the TIGERSearch tool. The following section introduces a corpus description language for syntactically annotated corpora which is used for both defining and querying a corpus. Afterwards, the architecture of the corpus query processor is presented and some aspects of the graphical user interface are described. Finally, plans for the future are presented and the results of the project are discussed.

2 Related work

Treebank encoding formats

Since the TIGERSearch tool works on treebanks, a treebank import format is needed. As it supports the required graph model of the TIGER corpus, the *Negra format* could be used (Brants 1997). Unfortunately, the format is limited to the features used in the Negra corpus – syntactic category for non-terminal nodes, word form, part-of-speech tag, and morphological description for terminal nodes. Further features such as lemma cannot be added.

The *bracketing format* is used in many treebank projects such as the Penn Treebank. Though only one feature can be encoded for both terminal and non-terminal nodes, any desired feature values can be combined as one value using a separator such as the `"-"` symbol (e.g. `PRELS-Masc.Nom.Sg.`, cf. token *der* in fig. 1). A drawback of this solution is that the individual feature values have to be isolated in further processing. In the context of the TIGER corpus, an additional feature has to encode the token position of the terminal nodes. Otherwise the original token order is lost in sentences that comprise crossing edges.

The XML instantiation of the *Corpus Encoding Standard* (XCES, cf. Ide et al. 2000) has been introduced to define a standard for linguistic annotation. The main idea of this standard is to use stand-off annotation, i.e. to encode each an-

notation level in a different file. The annotation levels of a corpus are merged by referencing the basic annotation level, usually the token level. The recommendations of the XCES initiative for encoding syntactic annotation take into account all known representations of syntactic annotations (Ide and Romary 2000a). Thus, phrase structures can be encoded by embedding elements, and graph structures are represented by references. However, defining the XCES format as the import format of the TIGERSearch tool has some technical drawbacks: Due to the flexibility of the XCES approach, the concrete realization of an encoding (separation of the annotation levels, representation of graph structure) is left to the corpus. Thus, a restricted version of the XCES standard would be necessary to define a unique import format.

The presented treebank encoding formats are interesting candidates for the required import format. Unfortunately, there are some drawbacks for each format. Thus, an alternative encoding will be developed within the next section. However, the approaches presented previously have greatly influenced the development process.

Treebank search tools

The first treebank search tool was the *tgrep* tool (Pito 1993) which was distributed with the first release of the Penn Treebank. The tgrep tool has been designed to process syntactic annotations that are based on trees. It uses the bracketing format as an input format. Before processing a new corpus, the corpus represented in bracketing format is first converted into a binary index. This index represents the corpus as a whole as well as information about the corpus that can improve query processing efficiency. As it is based on the use of regular expressions, the query language of tgrep is quite difficult to learn, especially for occasional usage.

The *ICECUP* tool has been designed as an alternative to tools using logical query languages such as tgrep or CorpusSearch (cf. Wallis and Nelson 2000). In ICECUP corpus queries are *drawn* by the user, i.e. nodes and relations between nodes can be expressed in a graphical query editor. ICECUP is based on a professional graphical user interface that includes graphical query input and graphical visualization of query results. Similar to tgrep, ICECUP makes use of an index-based approach that is limited to annotations of tree structures.

The *VIQTORYA* tool is the only tool that supports the data model used in the TIGER corpus. In a preprocessing step a treebank is transformed into a relational database. Afterwards, the transformed corpus can be accessed by SQL queries. For the sake of user-friendliness a logical query language has been developed. Queries expressed in this language are converted into SQL queries to access the corpus database. The main motivation of the database approach is to obviate the development of a query processing engine. One of its disadvantages

is the memory-intensive mapping of the graph structures to the relational database which might lead to a database size that cannot be handled. In addition, corpora that can be processed by VIQTORYA are currently limited to corpora annotated according to the Negra format.

The TIGERSearch tool tries to combine the interesting features of the presented tools. The definition of a user-friendly query language and development of a graphical user interface are the main focus.

3 A corpus description language

This section presents a survey of the corpus description language. A detailed introduction can be found in König and Lezius (2002a), a formal definition can be found in König and Lezius (2002b).

A treebank query tool usually defines two languages to communicate with the user. The *corpus definition language* is the encoding format that is used to define a new treebank. For example, tools such as tgrep use the bracketing format for corpus definition. The *corpus query language* lets the user express his queries to the corpus. These two languages usually differ greatly.

The *corpus description language* of TIGERSearch tries to avoid this separation. As an advantage the user has to learn just one language. Furthermore, the design of the two languages is much more appropriate as the definition language is a subset of the query language. The languages simply differ in the level of specification (totally specified vs. underspecified).

A walk through the description language

Nodes

Syntactic phrases usually comprise morphosyntactic information such as part-of-speech, case, number, etc. In computational linguistics, feature structures encode such linguistic data. To express this kind of information *feature records*, i.e. flat feature structures whose feature values are constants, are used. Boolean expressions can be used both on the feature value level and on the feature-value-pair level. For example, all proper nouns (`NE`) and nouns (`NN`) can be retrieved by the following query:

```
[pos= ("NE"|"NN")]
```

To search for the word form *Mann* which is used as a noun two feature-value-pairs are combined by conjunction:

```
[word="Mann & "pos="NN"]
```

Feature values which cannot be enumerated (for features such as *word* or *lemma*) can be referred to by regular expressions. For example, nouns with initial *H* can be described by (the "/" symbols mark a regular expression):

```
[word = /H.*/ & pos="NN"]
```

Node relations

Since graphs are two-dimensional objects, one basic node relation for each dimension is needed: direct precedence "." for the horizontal dimension and labelled direct dominance ">L" for the vertical dimension (the precedence of two inner nodes is defined as the precedence of their leftmost terminal successors, cf. König and Lezius 2002a). There are also several derived relations such as general dominance or a sibling relation. The following expression describes a noun phrase that comprises a relative clause (cf. extraposed relative clause in fig. 1):

```
[cat="NP"] >RC [cat="S"]
```

Graph descriptions

A graph description consists of restricted Boolean expressions of node relations. Negation is not allowed. Variables are used to express identity of two node descriptions. The following expression describes noun phrases (NP) that comprise a determiner (ART) and a noun (NN):

```
#np:[cat="NP"] > [pos="ART"]  &  #np > [pos="NN"]
```

Please note that the corpus graph in fig. 1 matches this expression. To restrict the noun phrase to a determiner and a noun only, the node predicate *arity* can be used:

```
#np:[cat="NP"]>[pos="ART"]  &  #np>[pos="NN"]  &  arity(#np,2)
```

Types

The user can also define type hierarchies for features. A type definition might include subtypes or constants. The following example illustrates a type hierarchy for the STTS tagset:

```
stts := nominal, verbal, others.
nominal := Noun, properNoun, pronoun.
noun := "NN".
properNoun := "NE".
pronoun := "PPPER", "PPOS", "PRELS", ... .
```

Such a hierarchy can be used to formulate queries in a more adequate way:

```
[pos=nominal] .* [pos="VVFIN"]
```

Corpus definition vs. corpus query

Any expression of the corpus description language represents a corpus query. In contrast, a corpus definition is restricted. A corpus is defined graph by graph and constitutes a corpus graph in a unique way: all features of a node are specified, node relations are exactly defined, only conjunctions may be used, etc. The following example defines the extraposed relative clause in fig. 1:

```
<sentence id="graphid">
 "t1":[word="der" & pos="PPRELS" & morph="Masc.Nom.Sg"] &
 "t2":[word="lacht" & pos="VVFIN" & morph="3.Sg.Pres.Ind"] &
 "n1":[cat="S"] &
 "t1" . "t2"  &  "n1" >SB "t1"  &  "n1" >HD "t2" &
 arity("n1",2) & root("n1")
</sentence>
```

The TIGER-XML format

The concept of merging corpus definition and corpus query into a common corpus description language has some obvious advantages. But there also some technical drawbacks: First, some symbols are difficult to encode as feature values (e.g. `word="""`). Second, corpus definitions cannot be validated. Third, the corpus description language does not allow the encoding of corpus query results combined with the matching corpus graphs.

For this reason an XML-based encoding format has been developed which is semantically equivalent to the corpus definition language. The so-called TIGER-XML format represents the graph structure by using references. By assigning XSLT stylesheets to TIGER-XML documents, conversions into different formats can be easily realized. The following example illustrates the XML encoding of the relative clause in fig. 1:

```
<s id="graphid">
  <graph root="n1">
    <terminals>
      <t id="t1" word="der"   pos="PRELS" morph="Masc.Nom.Sg" />
      <t id="t2" word="lacht" pos="VVFIN" morph="3.Sg.Pres.Ind" />
    </terminals>
    <nonterminals>
      <nt id="n1" cat="S">
```

```
        <edge label="SB" idref="t1" />
        <edge label="HD" idref="t2" />
      </nt>
    </nonterminals>
  </graph>
</s>
```

The TIGER-XML format is used as the default import format of the TIGERSearch tool. Corpora encoded in different formats have to be converted into TIGER-XML first. To support as many existing treebanks as possible, import filters, i.e. converters to TIGER-XML, for well-known treebank encoding formats are included in the TIGERSearch distribution.

4 Corpus query processing

As a step towards a query processor for the TIGERSearch query language, the TIGER calculus is defined. The calculus is a straightforward adaptation of the resolution calculus for Horn clauses in logic programming languages. Correctness and completeness of the calculus are shown in König and Lezius (2002b). Thus, any correct and complete implementation of the TIGER calculus is a correct and complete interpreter of the TIGERSearch query language.

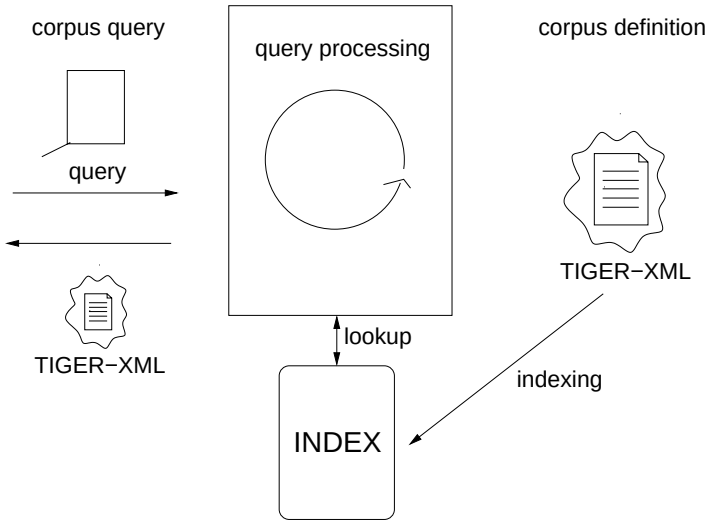


Figure 2: System architecture of TIGERSearch

System architecture

The implementation of the calculus developed in the TIGERSearch project is based on an index (cf. fig. 2). In a preprocessing step, a new corpus encoded

in the TIGER-XML format is converted into a binary index representation. The index represents the results of inference rules of the calculus that can be applied before processing a query. Thus, query processing at runtime consists of applying inference rules that are based on the facts represented in the index and the user's query. The query processing algorithm is described in the next section.

Query processing

The query processing algorithm is illustrated by an example. For the sake of clarity, this example presents a restricted view on the implemented algorithm. The following example query is processed on a corpus that only comprises the example sentence in fig. 1:

```
#p:[cat="NP" | cat="S"] > #t:[pos="PRELS"] & arity(#p,2)
```

The first step transforms the query into disjunctive normal form. All disjuncts are moved to the graph description level. The result of the normalization step is a list of independent disjuncts that can be processed separately:

```
( #p:[cat="NP"] & #t:[pos="PRELS"] & #p > #t & arity(#p,2) ) |
( #p:[cat="S"] & #t:[pos="PRELS"] & #p > #t & arity(#p,2) )
```

The following description concentrates on the second disjunct:

```
#p:[cat="S"] & #t:[pos="PRELS"] & #p > #t & arity(#p,2)
```

In a second step the node descriptions of the query disjunct are admitted to the so-called store. The store collects all variables and their constraints used in the query. As a result of the second step, the query is divided into a query skeleton and the store:

`#p` `& #t & #p > #t & arity(#p,2)`

variable	assignment	constraint
#p	--	[cat="S"]
#t	--	[pos="PRELS"]

In the third step all corpus graphs are checked for a match of the query. For each corpus graph the conjuncts of the query skeleton are traversed from left to right: Node variables such as `#p` are assigned to matching corpus graph nodes and node relations (`#p>#t`) or node predicates (`arity(#p,2)`) are checked. If an assignment leads to a logical clash or if a predicate or relation check is not successful, backtracking is initiated and processing continues with the last conjunct.

In the example shown, the algorithm starts with the conjunct `#p`. If the root node of the corpus graph in fig. 1 is assigned to the variable `#p`, the contraints `[cat="CS"]` of the root node and `[cat="S"]` in the store are unified. This unification leads to a logical clash. If `#p` is assigned to the `S`-node in the right subgraph of the corpus graph, the assignment is successful. The result of the unification is represented in the store:

`#p & #t & #p > #t & arity(#p,2)`

variable	assignment	constraint
#p	"n1"	[cat="S"]
#t	--	[pos="PRELS"]

In this case, the node variable `#t` has to be assigned next. `#t` can be successfully assigned to the token *der*. Note that the result of the unification of the stored constraint of `#t` and the feature-value-pairs of the corpus node *der* is the new constraint of `#t`:

`#p & #t & #p > #t & arity(#p,2)`

variable	assignment	constraint
#p	"n1"	[cat="S"]
#t	"t1"	[word="der" & pos="PRELS" & morph="Masc.Nom.Sg"]

The node relation and node predicate checks for the assigned corpus nodes of `#p` and `#t` are also successful. Now all conjuncts have been traversed – a match has been found. The match result is represented by the current store:

`#p & #t & #p > #t & arity(#p,2)` $\surd$

variable	assignment	constraint
#p	"n1"	[cat="S"]
#t	"t1"	[word="der" & pos="PRELS" & morph="Masc.Nom.Sg"]

Efficient query processing

The backtracking algorithm realizes an exhaustive search on all variable assignments which leads to correct and complete, but sometimes inefficient query processing. To increase efficiency, query optimization and search space filter strategies have been applied. The following example query will illustrate the query optimization strategy:

```
#p:[cat="S"] & #t:[pos="PPER"] & #p > #t & arity(#p,2)
```

The traversal of the query conjuncts from left to right is inefficient in this example. As unification of corpus nodes with node descriptions in the query are much more time-consuming than node relation or node predicate checks, the query should be reorganized in the following way:

```
#p:[cat="S"] & arity(#p,2) & #t:[pos="PPER"] & #p > #t
```

This query variant is semantically equivalent to the original query. The query optimization in this example increases efficiency by 75%.

The main idea of search space filters is to reduce the set of corpus graphs that are match candidates for a query. As a first step forward, a node filter has been implemented. It concentrates on the node descriptions of a query and ignores node relations and node predicates. It analyzes the node descriptions and reduces the original query to a conjunction of restrictive node descriptions. For example, the query shown above would be reduced to:

```
#t:[pos="PPER"]
```

Note that the reduced query is a generalization of the original query. Corpus graphs that match the reduced query are match candidates for the original query. Corpus graphs that do not match the reduced query can be ignored. If the example query is processed on the TIGER corpus, the set of corpus graph candidates is reduced to 25%.

5 The graphical user interface

Tools such as ICECUP show that a graphical user interface is of particular importance for a corpus query tool. Thus, a sophisticated user interface has been developed for the TIGERSearch software. The TIGERGraphViewer visualizes the results of a corpus query. Users can browse through the matching corpus graphs and export their favourite matches as TIGER-XML documents or interactive SVG images.

To assist new users of the TIGERSearch software, a graphical query editor called *TIGERin* has been developed (cf. Voormann 2002). Queries can be drawn by combining node and node relation objects (cf. fig. 3). The user can concentrate on the corpus and does not have to first get involved in the logical query language.

6 Conclusion

This paper has presented the concept of the treebank search tool TIGERSearch. The software has been implemented in Java and is available for all major platforms. Current work concentrates on the improvement of query processing efficiency by realizing more complex search space filters. On a technical level, a

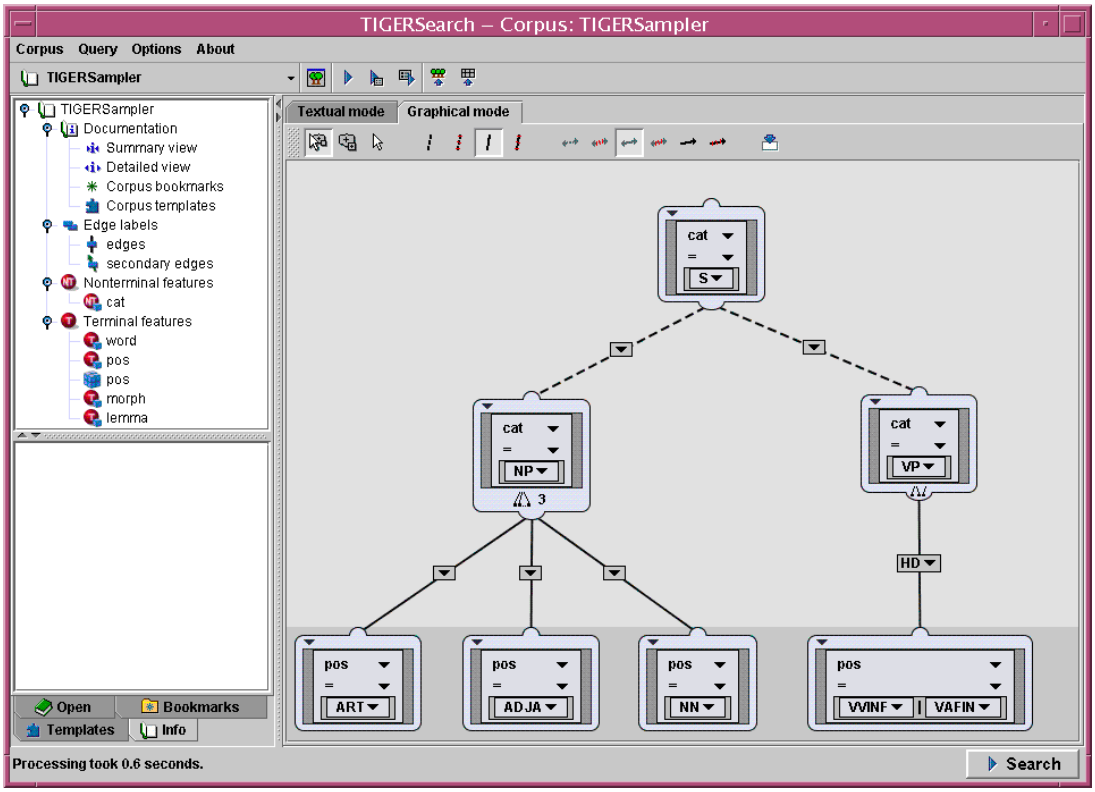


Figure 3: The graphical query editor

client/server architecture will be implemented. Thin clients will communicate with a high-end server that serves incoming query requests.

This scenario is also interesting for the publication of the TIGER corpus. Registered users of the TIGER corpus will be able to access the corpus with any browser or applet client.

TIGERSearch offers an expressive query language and efficient query evaluation – at least for modestly-sized corpora. For larger corpora, more effective indexing and search space filter strategies have yet to be realized. The graphical user interface, especially the graphical query editor and the visualization of query results, makes TIGERSearch a powerful and easy to handle search tool.

Literaturverzeichnis

1 Abiteboul et al. 2000 ABITEBOUL, Serge ; BUNEMAN, Peter ; SUCIU, Dan: *Data on the Web: from Relations to Semistructured Data and XML*. Morgan Kaufmann, 2000

2 Albert et al. 2002 ALBERT, Stefanie ; ANDERSSEN, Jan ; BRACHT, Tobias ; BRANTS, Sabine ; BRANTS, Thorsten ; DIPPER, Stefanie ; EISENBERG, Peter ; LANGNER, Robert ; PLAEHN, Oliver ; PREIS, Cordula ; PUSSEL, Marcus ; SCHRADER, Bettina ; SCHWARTZ, Anne ; SMITH, George ; USZKOREIT, Hans: Das TIGER Annotationsschema / Universität des Saarlandes, Universität Stuttgart, Universität Potsdam. 2002. – Forschungsbericht

3 ANC 2001 *American National Corpus (ANC). Website*. 2001. – URL `http://www.cs.vassar.edu/~ide/anc/`

4 Apparao et al. 1998 APPARAO, Vidur ; BYRNE, Steve ; CHAMPION, Mike ; ISAACS, Scott ; JACOBS, Ian ; HORS, Arnaud L. ; NICOL, Gavin ; ROBIE, Jonathan ; SUTOR, Robert ; WILSON, Chris ; WOOD, Lauren: Document Object Model (DOM) Level 1 Specification / W3C. URL `http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/`, 1998. – W3C Recommendation

5 Argenton 1998 ARGENTON, Hans: *Indexierung und Retrieval von Feature-Bäumen am Beispiel der linguistischen Analyse von Textkorpora*. Sankt Augustin : infix-Verlag, 1998. – Dissertation der Universität Tübingen

6 Baeza-Yates und Gonnet 1996 BAEZA-YATES, Ricardo ; GONNET, Gaston H.: Fast Text Searching for Regular Expressions or Automaton Simulation over Tries. In: *Journal of ACM* 6 (1996), Nr. 43, S. 915–936

7 Baeza-Yates und Ribeiro-Neto 1999 BAEZA-YATES, Ricardo ; RIBEIRO-NETO, Berthier: *Modern Information Retrieval*. Addison Wesley, ACM Press, 1999

8 Barlow 2002 BARLOW, Michael: *Corpus Linguistics. Link Collection*. 2002. – URL `http://www.ruf.rice.edu/~barlow/corpus.html`

9 Barnard und Ide 1997 BARNARD, David T. ; IDE, Nancy: The Text Encoding Initiative: Flexible and Extensible Document Encoding. In: *Journal of the American Society for Information Science* 48 (1997), Nr. 7, S. 622–628

10 Bird et al. 2000 BIRD, Steven ; BUNEMAN, Peter ; TAN, Wan-Chiew: Towards a query language for annotation graphs. In: *Second International Conference on Language Resources and Evaluation (LREC 2000)*. Athen, 2000, S. 807–814

11 Bird und Liberman 2001 BIRD, Steven ; LIBERMAN, Mark: A formal framework for linguistic annotation. In: *Speech Communication* 33 (2001), Nr. 1,2, S. 23–60

12 Bird et al. 2002 BIRD, Steven ; MAEDA, Kazuaki ; MA, Xiaoyi ; LEE, Haejoong ; RANDALL, Beth ; ZAYAT, Salim: TableTrans, MultiTrans, InterTrans and TreeTrans: Diverse Tools Built on the Annotation Graph Toolkit. In: *Proceedings of the Third Conference on Language Resources and Evaluation (LREC 2002)*. Las Palmas, 2002, S. 364–370

13 Blackburn et al. 1993 BLACKBURN, Patrick ; GARDENT, Claire ; MEYER-VIOL, Wilfried: Talking About Trees. In: *Proceedings of the 6th Conference of the European Chapter of the Association for Computational Linguistics (EACL 1993)*. Utrecht, 1993, S. 21–29

14 BNC 2001 *British National Corpus (BNC). Website*. 2001. – URL `http://info.ox.ac.uk/bnc/`

15 Boag et al. 2002 BOAG, Scott ; CHAMBERLIN, Don ; FERNANDEZ, Mary F. ; FLORESCU, Daniela ; ROBIE, Jonathan ; SIMÉON, Jérôme ; STEFANESCU, Mugur: XQuery 1.0: An XML Query Language / W3C. URL `http://www.w3.org/TR/xquery/`, November 2002. – W3C Working Draft

16 Bouma und Kloostermann 2002 BOUMA, Gosse ; KLOOSTERMANN, Geert: Querying Dependency Treebanks in XML. In: *Proceedings of the Third Conference on Language Resources and Evaluation (LREC 2002)*. Las Palmas, 2002, S. 1686–1691

17 Brants et al. 2002 BRANTS, Sabine ; DIPPER, Stefanie ; HANSEN, Silvia ; LEZIUS, Wolfgang ; SMITH, George: The TIGER Treebank. In: *Proceedings of the Workshop on Treebanks and Linguistic Theories*. Sozopol, 2002

18 Brants und Hansen 2002 BRANTS, Sabine ; HANSEN, Silvia: Developments in the TIGER Annotation Scheme and their Realization in the Corpus. In: *Proceedings of the Third Conference on Language Resources and Evaluation (LREC 2002)*. Las Palmas, 2002, S. 1643–1649

19 Brants 1997 BRANTS, Thorsten: The Negra Export Format for Annotated Corpora, Version 3 / Institut für Computerlinguistik, Universität des Saarlandes. 1997. – Forschungsbericht

20 Brants et al. 1997 BRANTS, Thorsten ; HENDRIKS, Roland ; KRAMP, Sabine ; KRENN, Brigitte ; PREIS, Cordula ; SKUT, Wojciech ; USZKOREIT, Hans: Das Negra-Annotationsschema / Institut für Computerlinguistik, Universität des Saarlandes. 1997. – Forschungsbericht

21 Brants und Plaehn 2000 BRANTS, Thorsten ; PLAEHN, Oliver: Interactive Corpus Annotation. In: *Proceedings of the Second International Conference on Language Resources and Engineering (LREC 2000)*. Athen, 2000, S. 453–459

22 Brants et al. 1999 BRANTS, Thorsten ; SKUT, Wojciech ; USZKOREIT, Hans: Syntactic Annotation of a German Newspaper Corpus. In: *Proceedings of the ATALA Treebank Workshop*. Paris, 1999, S. 69–76

23 Bray et al. 2000 BRAY, Tim ; PAOLI, Jean ; SPERBERG-MCQUEEN, C.M. ; MALER, Eve: Extensible Markup Language (XML) 1.0 (Second Edition) / W3C. URL `http://www.w3.org/XML/`, 2000. – W3C Recommendation

24 Brew 1999 BREW, Chris: An extensible visualization tool to aid treebank exploration. In: USZKOREIT, Hans (Hrsg.): *Proceedings of the First Workshop on Linguistically Interpreted Corpora (LINC 1999)*. Bergen, 1999, S. 49–55

25 Brill 1995 BRILL, Eric: Unsupervised learning of disambiguation rules for part-of-speech tagging. In: YAROWSKY, David ; CHURCH, Kenneth W. (Hrsg.): *Proceedings of the Third Workshop on Very Large Corpora*. Cambridge, 1995, S. 1–13

26 Burnard 1996 BURNARD, Lou: Introducing SARA: exploring the BNC with SARA. In: *Proceedings of the Second Conference on Teaching and Learner Corpora (TALC 1996)* Lancaster University, 1996

27 Cahill und van Genabith 2002 CAHILL, Aoife ; GENABITH, Josef van: TTS – A Treebank Tool Suite. In: *Proceedings of the Third Conference on Language Resources and Evaluation (LREC 2002)*. Las Palmas, 2002, S. 1712–1717

28 Calder 1999 CALDER, Jo: *Interabora Tree Delivery Service - Website*. 1999. – URL `http://www.ltg.ed.ac.uk/~jo/interarbora/`

29 Calder 2000a CALDER, Jo: Interarbora and Thistle - Delivering linguistic structure by the Internet. In: *Proceedings of the Second International Conference on Language Resources and Engineering (LREC 2000)*. Athen, 2000, S. 1163–1166

30 Calder 2000b CALDER, Jo: Thistle and Interarbora. In: *Proceedings of the 18th International Conference on Computational Linguistics (COLING 2000)*. Saarbrücken, 2000, S. 992–996

31 Carletta et al. 2002 CARLETTA, Jean ; MCKELVIE, David ; ISARD, Amy: *Supporting linguistic annotation using XML and stylesheets*. In: SAMPSON, Geoffrey ; MCCARTHY, D. (Hrsg.): *Readings in Corpus Linguistic*, Continuum International, 2002

32 Chamberlin et al. 2001 CHAMBERLIN, Don ; FANKHAUSER, Peter ; MARCHIORI, Massimo ; ROBIE, Jonathan: XML Query Requirements / W3C. URL `http://www.w3.org/TR/xmlquery-req`, 2001. – W3C Working Draft

33 Christ 1994 CHRIST, Oliver: A modular and flexible architecture for an integrated corpus query system. In: *Proceedings of the Third Conference on Computational Lexicography and Text Research (COMPLEX 1994)*. Budapest, 1994, S. 23–32

34 Christ et al. 1999 CHRIST, Oliver ; SCHULZE, Bruno M. ; HOFMANN, Anja ; KÖNIG, Esther: *Corpus Query Processor (CQP). User's Manual*. Institut für Maschinelle Sprachverarbeitung, Universität Stuttgart, 1999

35 Church 1988 CHURCH, Kenneth W.: A Stochastic Parts Program and Noun Phrase Parser for Unrestricted Text. In: *Second Conference on Applied Natural Language Processing (ANLP 1988)*. Austin, 1988, S. 136–143

36 Clark 1999 CLARK, James: XSL Transformations (XSLT) 1.0 / W3C. URL `http://www.w3.org/TR/xslt`, 1999. – W3C Recommendation

37 Clark und DeRose 1999 CLARK, James ; DEROSE, Steve: XPath 1.0 / W3C. URL `http://www.w3.org/TR/xpath`, 1999. – W3C Recommendation

38 Cotton und Bird 2002 COTTON, Scott ; BIRD, Steven: An integrated framework for treebanks and multilayer annotations. In: *Proceedings of the Third Conference on Language Resources and Evaluation (LREC 2002)*. Las Palmas, 2002, S. 1670–1677

39 Cutting et al. 1992 CUTTING, D. ; KUPIEC, J. ; PETERSEN, J. ; SIBUN, P.: A practical part-of-speech tagger. In: *Proceedings of the Third Conference on Applied Natural Language Processing (ANLP 1992)*. Trento, 1992, S. 133–140

40 DEREKO 2001 *Deutsches Referenzkorpus (DEREKO). Website*. 2001. – URL `http://www.sfs.nphil.uni-tuebingen.de/dereko/`

41 Deutsch et al. 1998 DEUTSCH, Alin ; FERNANDEZ, Mary ; FLORESCU, Daniela ; LEVY, Alon ; SUCIU, Dan: XML-QL: A Query Language for XML / W3C. URL `http://www.w3.org/TR/NOTE-xml-ql/`, 1998. – W3C Note

42 Docherty und Heid 1998 DOCHERTY, Vincent J. ; HEID, Ulrich: Computational Metalexicography in Practice – Corpus-based support for the revision of a commercial dictionary. In: *Proceedings of the 8th Euralex International Congress*. Liège, 1998, S. 333 – 346

43 Dörre 1996 DÖRRE, Jochen: *Feature-Logik und Semiunifikation*. Sankt Augustin : infix-Verlag, 1996. – Dissertation der Universität Stuttgart

44 Dörre und Dorna 1993 DÖRRE, Jochen ; DORNA, Michael: CUF - A Formalism for Linguistic Knowledge Representation / DYANA 2. August 1993 (R.1.2A). – Forschungsbericht

45 Dörre et al. 1996 DÖRRE, Jochen ; GABBAY, Dov M. ; KÖNIG, Esther: Fibred Semantics for Feature-based Grammar Logic. In: *Journal of Logic, Language, and Information. Special Issue on Language and Proof Theory* 5 (1996), S. 387–422

46 Duchier und Niehren 1999 DUCHIER, Denys ; NIEHREN, Joachim: Solving Dominance Constraints with Finite Set Constraint Programming / Universität des Saarlandes, Programming Systems Lab. 1999. – Forschungsbericht

47 Emele 1997 EMELE, Martin: *Der TFS-Repräsentationsformalismus*. 1997. – Dissertation der Universität Stuttgart

48 Eschweiler 2001a ESCHWEILER, Sebastian: DBDOM - Eine XML-DOM-Implementierung in SQL. In: *JavaMagazin 11/2001* (2001), S. 87–92

49 Eschweiler 2001b ESCHWEILER, Sebastian: dbXML - Eine XML-basierte Datenbank. In: *JavaMagazin 9/2001* (2001), S. 98–103

50 Evert 2001 EVERT, Stefan: *CQP Query Language Tutorial*. Institut für Maschinelle Sprachverarbeitung, Universität Stuttgart, 2001. – URL `http://www.ims.uni-stuttgart.de/projekte/CorpusWorkbench/`

51 Evert und Fitschen 2001 EVERT, Stefan ; FITSCHEN, Arne: Textkorpora. In: CARSTENSEN, Kai-Uwe ; EBERT, Christian ; ENDRISS, Cornelia ; JEKAT, Susanne ; KLABUNDE, Ralf ; LANGER, Hagen (Hrsg.): *Computerlinguistik und Sprachtechnologie - Eine Einführung*. Heidelberg, Berlin : Spektrum Akademischer Verlag, 2001, S. 369–376

52 Evert und Kermes 2002 EVERT, Stefan ; KERMES, Hannah: YAC - A Recursive Chunker for Unrestricted German Text. In: *Proceedings of the Third Conference on Language Resources and Evaluation (LREC 2002)*. Las Palmas, 2002, S. 1805–1812

53 Fallside 2001 FALLSIDE, David C.: XML Schema Part 0: Primer / W3C. URL `http://www.w3.org/TR/xmlschema-0/`, 2001. – W3C Recommendation

54 Ferraiolo 2001 FERRAIOLO, Jon: Scalable Vector Graphics (SVG) 1.0 Specification / W3C. URL `http://www.w3.org/TR/SVG/`, 2001. – W3C Recommendation

55 Fitschen und Lezius 2001a FITSCHEN, Arne ; LEZIUS, Wolfgang: Eine Einführung in XML und XSLT (Teil 1). In: *JavaMagazin 9/2001* (2001), S. 93–97

56 Fitschen und Lezius 2001b FITSCHEN, Arne ; LEZIUS, Wolfgang: Eine Einführung in XML und XSLT (Teil 2). In: *JavaMagazin 10/2001* (2001), S. 83–87

57 Garside et al. 1987 GARSIDE, Roger ; SAMPSON, Geoffrey ; LEECH, Geoffrey (Hrsg.): *The Computational Analysis of English: a corpus-based approach*. London : Longman, 1987

58 Goldman et al. 1999 GOLDMAN, Roy ; MCHUGH, Jason ; WIDOM, Jennifer: From Semistructured Data to XML: Migrating the Lore Data Model and Query Language. In: *Proceedings of the 2nd International Workshop on the Web and Databases (WebDB 1999)*. Philadelphia, 1999, S. 25–30

59 Gorn 1967 GORN, Saul: Explicit definitions and linguistic dominoes. In: HART, John F. ; TAKASU, Satoru (Hrsg.): *Systems and Computer Science - Proceedings of the Conference held at University of Western Ontario, 1965*, University of Toronto Press, 1967, S. 77–115

60 Green und Rubin 1977 GREEN, Barbara B. ; RUBIN, Gerald M.: Automatic Grammatical Tagging of English / Department of Linguistics, Brown University. Providence, Rhode Island, 1977. – Forschungsbericht

61 Greenbaum 1996 GREENBAUM, Sidney: *Comparing English Worldwide: The International Corpus of English*. Clarendon Press, 1996

62 Hajic 1999 HAJIC, Jan: Building a Syntactically Annotated Corpus: The Prague Dependency Treebank. In: HAJICOVA, Eva (Hrsg.): *Issues of Valency and Meaning. Studies in Honour of Jarmila Panevova*. Charles University Press, 1999, S. 106–132

63 van Halteren 1997 HALTEREN, Hans van: *Excursions into Syntactic Databases*. Rodopi, 1997

64 van Halteren et al. 1998 HALTEREN, Hans van ; ZAVREL, F. ; DAELEMANS, W.: Improving Data Driven Wordclass Tagging by System Combination. In: *Proceedings of the COLING-ACL 1998*, 1998, S. 491–497

65 Heid et al. 2000 HEID, Ulrich ; EVERT, Stefan ; DOCHERTY, Vincent ; WORSCH, Wolfgang ; WERMKE, Matthias: A data collection for semi-automatic corpus-based updating of dictionaries. In: *Proceedings of the 9th EURALEX International Congress*. Stuttgart, 2000, S. 183 – 195

66 Heid und Mengel 1999 HEID, Ulrich ; MENGEL, Andreas: Query Language for Research in Phonetics. In: *Proceedings of the International Congress of Phonetic Sciences (ICPhS 1999)*. San Francisco, 1999, S. 1225–1228

67 Hinrichs et al. 2000 HINRICHS, Erhard W. ; BARTELS, Juli ; KAWATA, Yasuhiro ; KORDONI, Valia ; TELLJOHANN, Heike: The Verbmobil Treebanks. In: ZÜHLKE, Werner ; SCHUKAT-TALAMAZZINI, Ernst G. (Hrsg.): *Konvens 2000 Sprachkommunikation*, VDE-Verlag, 2000, S. 107–112

68 Höhfeld und Smolka 1988 HÖHFELD, Markus ; SMOLKA, Gert: Definite Relations over Constraint Languages / IBM Deutschland. Stuttgart, 1988. – Forschungsbericht

69 Ide 1998 IDE, Nancy: Encoding Linguistic Corpora. In: *Proceedings of the Sixth Workshop on Very Large Corpora*. Montreal, 1998, S. 9–17

70 Ide 2000a IDE, Nancy: The XML framework and its implications for corpus access and use. In: *Proceedings of Data Architectures and Software Support for Large Corpora*. Paris, 2000, S. 28–32

71 Ide 2000b IDE, Nancy: The XML framework and its implications for the development of natural language processing tools. In: *Proceedings of the Workshop on Using Toolsets and Architectures to Build NLP Systems*. Luxemburg, 2000

72 Ide et al. 2000 IDE, Nancy ; BONHOMME, Patrice ; ROMARY, Laurent: XCES: An XML-based encoding standard for linguistic corpora. In: *Proceedings of the Second International Conference on Language Resources and Engineering (LREC 2000)*. Athen, 2000, S. 825–830

73 Ide und Brew 2000 IDE, Nancy ; BREW, Chris: Requirements, Tools, and Architectures for Annotated Corpora. In: *Proceedings of Data Architectures and Software Support for Large Corpora*. Paris, 2000, S. 1–5

74 Ide und Romary 2000a IDE, Nancy ; ROMARY, Laurent: Encoding syntactic annotation. In: ABEILLÉ, Anne (Hrsg.): *Building and using syntactically annoted corpora*. Kluwer, 2000

75 Ide und Romary 2000b IDE, Nancy ; ROMARY, Laurent: XML support for annotated language resources. In: *Workshop on Web-Based Language Documentation and Description*. Philadelphia, 2000, S. 148–153

76 Ide und Romary 2001 IDE, Nancy ; ROMARY, Laurent: A Common Framework for Syntactic Annotation. In: *Proceedings of ACL 2001*. Toulouse, 2001, S. 298–305

77 Jansen 2002 JANSEN, Rudolf: Große Koalition - Integration von XML-Technologien in Oracle 9i. In: *JavaMagazin 4/2002* (2002), S. 88–93

78 Kallmeyer 2000 KALLMEYER, Laura: A query tool for syntactically annotated corpora. In: *Proceedings of Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora*. Hongkong, 2000, S. 190–198

79 Kallmeyer und Steiner 2002 KALLMEYER, Laura ; STEINER, Ilona: Querying treebanks of spontaneous speech with VIQTORYA. In: *Traitement automatique des langues* 43 (2002), Nr. 2

80 König und Lezius 2000 KÖNIG, Esther ; LEZIUS, Wolfgang: A description language for syntactically annotated corpora. In: *Proceedings of the 18th International Conference on Computational Linguistics (COLING 2000)*. Saarbrücken, 2000, S. 1056–1060

81 König und Lezius 2002a KÖNIG, Esther ; LEZIUS, Wolfgang: The TIGER language - A Description Language for Syntax Graphs. Part 1: User's Guidelines. / Institut für Maschinelle Sprachverarbeitung, Universität Stuttgart. 2002. – Forschungsbericht

82 König und Lezius 2002b KÖNIG, Esther ; LEZIUS, Wolfgang: The TIGER language - A Description Language for Syntax Graphs. Part 2: Formal Definition. / Institut für Maschinelle Sprachverarbeitung, Universität Stuttgart. 2002. – Forschungsbericht

83 Kolb 2001 KOLB, Peter: Graphentheorie und Merkmalsstrukturen. In: CARSTENSEN, Kai-Uwe ; EBERT, Christian ; ENDRISS, Cornelia ; JEKAT, Susanne ; KLABUNDE, Ralf ; LANGER, Hagen (Hrsg.): *Computerlinguistik und Sprachtechnologie - Eine Einführung*. Heidelberg, Berlin : Spektrum Akademischer Verlag, 2001, S. 87–106

84 Leech et al. 1983 LEECH, G. ; GARSIDE, R. ; ATWELL, E.: The Automatic Grammatical Tagging of the LOB Corpus. In: *ICAME News* 7 (1983), S. 13–33

85 Leech und Eyes 1997 LEECH, Geoffrey ; EYES, Elizabeth: Syntactic Annotation: Treebanks. In: GARSIDE, Roger ; LEECH, Geoffrey ; MCENERY, Tony (Hrsg.): *Corpus annotation: linguistic information from computer text corpora*. New York : Addison Wesley Longman, 1997, Kap. 3, S. 34–52

86 Lezius 2001 LEZIUS, Wolfgang: Baumbanken. In: CARSTENSEN, Kai-Uwe ; EBERT, Christian ; ENDRISS, Cornelia ; JEKAT, Susanne ; KLABUNDE, Ralf ; LANGER, Hagen (Hrsg.): *Computerlinguistik und Sprachtechnologie - Eine Einführung*. Heidelberg, Berlin : Spektrum Akademischer Verlag, 2001, S. 377–385

87 Lezius 2002 LEZIUS, Wolfgang: TIGERSearch – Ein Suchwerkzeug für Baumbanken. In: BUSEMANN, Stephan (Hrsg.): *Proceedings der 6. Konferenz zur Verarbeitung natürlicher Sprache (KONVENS 2002)*. Saarbrücken, 2002, S. 107–114

88 Lezius 2003 LEZIUS, Wolfgang: Auf den Weg gebracht – Die XML-Anfragesprache XQuery im Praxiseinsatz. In: *JavaMagazin* (2003). – Angenommen zur Veröffentlichung

89 Lezius et al. 2002a LEZIUS, Wolfgang ; BIESINGER, Hannes ; GERSTENBERGER, Ciprian: *TIGER-XML Quick Reference Guide*. Institut für Maschinelle Sprachverarbeitung, Universität Stuttgart, 2002

90 Lezius et al. 2002b LEZIUS, Wolfgang ; BIESINGER, Hannes ; GERSTENBERGER, Ciprian: *TIGERRegistry Manual*. Institut für Maschinelle Sprachverarbeitung, Universität Stuttgart, 2002

91 Lezius et al. 2002c LEZIUS, Wolfgang ; BIESINGER, Hannes ; GERSTENBERGER, Ciprian: *TIGERSearch Manual*. Institut für Maschinelle Sprachverarbeitung, Universität Stuttgart, 2002

92 Lezius et al. 1998 LEZIUS, Wolfgang ; RAPP, Reinhard ; WETTLER, Manfred: A Freely Available Morphological Analyzer, Disambiguator, and Context Sensitive Lemmatizer for German. In: *Proceedings of the COLING-ACL 1998*. Montreal, 1998, S. 743–747

93 Marcus et al. 1994 MARCUS, Mitchell ; KIM, Grace ; MARCINKIEWICZ, Mary A. ; MACINTYRE, Robert ; BIES, Ann ; FERGUSON, Mark ; KATZ, Karen ; SCHASBERGER, Britta: The Penn Treebank: Annotating predicate argument structure. In: *Proceedings of the ARPA Human Language Technology Workshop*. Plainsboro, 1994, S. 114–119

94 Marcus et al. 1993 MARCUS, Mitchell ; SANTORINI, Beatrice ; MARCINKIEWICZ, Mary A.: Building a large annotated corpus of English: the Penn Treebank. In: *Computational Linguistics* 19 (1993), S. 313–330

95 McConnell 1993 MCCONNELL, Steve: *Code Complete - A practical handbook of software construction*. Microsoft Press, 1993

96 McHugh et al. 1998 MCHUGH, Jason ; WIDOM, Jennifer ; ABITEBOUL, Serge ; LUO, Qingshan ; RAJARAMAN, Anand: Indexing semistructured data / Stanford University Database Group. 1998. – Forschungsbericht

97 McHugh 2000 MCHUGH, Jason G.: *Data Management and Query Processing for Semistructured Data*, Stanford University Database Group, Dissertation, 2000

98 McHugh und Widom 1999 MCHUGH, Jason G. ; WIDOM, Jennifer: Query Optimization for XML. In: *Proceedings of the Twenty-Fifth International Conference on Very Large Databases*. Edinburgh, 1999, S. 315–326

99 McKelvie et al. 2001 MCKELVIE, David ; ISARD, Amy ; MENGEL, Andreas ; MØLLER, Morten B. ; GROSSE, Michael ; KLEIN, Marion: The MATE Workbench - An annotation tool for XML coded speech corpora. In: *Speech Communication* 33 (2001), Nr. 1-2, S. 97–112

100 Mengel und Lezius 2000 MENGEL, Andreas ; LEZIUS, Wolfgang: An XML-based encoding format for syntactically annotated corpora. In: *Proceedings of the Second International Conference on Language Resources and Engineering (LREC 2000)*. Athen, 2000, S. 121–126

101 Meurers 2003 MEURERS, Detmar: *On the use of electronic corpora for theoretical linguistics - Case studies from the syntax of German*. 2003. – Erscheint in: Lingua

102 Meuss und Strohmaier 1999a MEUSS, Holger ; STROHMAIER, Christian: A Filter for Structured Document Retrieval / CIS, LMU München. 1999. – Forschungsbericht

103 Meuss und Strohmaier 1999b MEUSS, Holger ; STROHMAIER, Christian: Improving Index Structures for Structured Document Retrieval. In: *Proceedings of the 21st Annual Colloquium on IR Research (IRSG 1999)*, 1999

104 Mírovský et al. 2002 MÍROVSKÝ, Jiří ; ONDRUŠKA, Roman ; PRŮŠA, Daniel: Searching through the Prague Dependency Treebank – Conception and Architecture. In: *Proceedings of the Workshop on Treebanks and Linguistic Theories*. Sozopol, 2002

105 Monson-Haefel 2000 MONSON-HAEFEL, Richard: *Enterprise JavaBeans*. 2nd edition. O'Reilly, 2000

106 Nelson et al. 2002 NELSON, Gerald ; WALLIS, Sean ; AARTS, Bas: *Exploring Natural Language: Working with the British Component of the International Corpus of English*. Amsterdam : John Benjamins, 2002 (Varieties of English Around the World G29)

107 Oaks und Wong 1999 OAKS, Scott ; WONG, Henry: *Java Threads*. 2nd edition. O'Reilly, 1999

108 Ottmann und Widmayer 1993 OTTMANN, Thomas ; WIDMAYER, Peter: *Algorithmen und Datenstrukturen*. Zweite Auflage. BI Wissenschaftsverlag, 1993

109 Pito 1993 PITO, Richard: *Documentation for tgrep*. LDC, University of Pennsylvania, 1993

110 Plaehn 2000 PLAEHN, Oliver: Annotates Zeichenroutine / Institut für Computerlinguistik, Universität des Saarlandes. 2000. – Forschungsbericht

111 Plaehn und Brants 2000 PLAEHN, Oliver ; BRANTS, Thorsten: Annotate – An Efficient Interactive Annotation Tool. In: *Proceedings of the Sixth Conference on Applied Natural Language Processing (ANLP 2000)*. Seattle, 2000

112 Randall 2000 RANDALL, Beth: *CorpusSearch User Manual*. University of Pennsylvania, 2000

113 Rapp und Lezius 2001 RAPP, Reinhard ; LEZIUS, Wolfgang: Statistische Wortartenannotierung für das Deutsche. In: *Sprache und Datenverarbeitung* 25 (2001), Nr. 2

114 Robie et al. 1998 ROBIE, Jonathan ; LAPP, Joe ; SCHACH, David: XML Query Language (XQL) / W3C. URL `http://www.w3.org/TandS/QL/QL98/pp/xql.html`, 1998. – W3C Proposal

115 Rogers und Vijay-Shanker 1992 ROGERS, James ; VIJAY-SHANKER, K.: Reasoning with Descriptions of Trees. In: *Proceedings of the Annual Meeting of the Association for Computational Linguistics*. Newark, Delaware, 1992

116 Rohde 2001 ROHDE, Douglas L. T.: *Tgrep2 User Manual*. School of Computer Science, Carnegie Mellon University, 2001

117 Rooth und Christ 1994 ROOTH, Mats ; CHRIST, Oliver: *A tutorial on corpus hacking. Part I. Corpora, data structures, algorithms, tools*. Präsentiert auf: Fourth Conference on Applied Natural Language Processing (ANLP 1994). 1994

118 Sampson 1993 SAMPSON, Geoffrey: The SUSANNE Corpus. In: *ICAME Journal* 17 (1993), S. 125–127

119 Sampson 1995 SAMPSON, Geoffrey: *English for the Computer: The SUSANNE Corpus and Analytic Scheme*. Clarendon Press, 1995

120 Schurtz 2002 SCHURTZ, Thomas: Erweiterung des VIQTORYA-Systems um Disjunktionen / Seminar für Sprachwissenschaft, Universität Tübingen. 2002. – Studienarbeit

121 Skut et al. 1998 SKUT, Wojciech ; BRANTS, Thorsten ; KRENN, Brigitte ; USZKOREIT, Hans: A Linguistically Interpreted Corpus of German Newspaper Texts. In: *Proceedings of the ESSLLI Workshop on Recent Advances in Corpus Annotation*. Saarbrücken, 1998, S. 18–24

122 Skut et al. 1997 SKUT, Wojciech ; KRENN, Brigitte ; BRANTS, Thorsten ; USZKOREIT, Hans: An Annotation Scheme for Free Word Order Languages. In: *Proceedings of the Fifth Conference on Applied Language Processing (ANLP 1997)*. Washington, 1997, S. 27–28

123 Steiner und Kallmeyer 2002 STEINER, Ilona ; KALLMEYER, Laura: VIQTORYA - A visual query tool for syntactically annotated corpora. In: *Proceedings of the Third Conference on Language Resources and Evaluation (LREC 2002)*. Las Palmas, 2002, S. 1704–1711

124 Taylor et al. 2000 TAYLOR, Ann ; MARCUS, Mitchell ; SANTORINI, Beatrice: *The Penn Treebank: An Overview*. In: ABEILLÉ, Anne (Hrsg.): *Building and using syntactically annoted corpora*, Kluwer Academic Press, 2000 (Language and Speech series)

125 Thielen et al. 1999 THIELEN, Christine ; SCHILLER, Anne ; TEUFEL, Simone ; STÖCKERT, Christine: Guidelines für das Tagging deutscher Textkorpora mit STTS / Institut für Maschinelle Sprachverarbeitung, Universität Stuttgart und Seminar für Sprachwissenschaft, Universität Tübingen. URL `http://www.ims.uni-stuttgart.de/projekte/corplex/TagSets/`, 1999. – Forschungsbericht

126 TIGER Projekt 2002 TIGER PROJEKT: *Website*. 2002. – URL `http://www.ims.uni-stuttgart.de/projekte/TIGER`

127 Voormann 2002 VOORMANN, Holger: TIGERin - Grafische Eingabe von Suchanfragen in TIGERSearch / Fakultät Informatik, Universität Stuttgart. 2002. – Diplomarbeit

128 Voormann und Lezius 2002 VOORMANN, Holger ; LEZIUS, Wolfgang: TIGERin - Grafische Eingabe von Benutzeranfragen für ein Baumbank-Anfragewerkzeug. In: BUSEMANN, Stephan (Hrsg.): *Proceedings der 6. Konferenz zur Verarbeitung natürlicher Sprache (KONVENS 2002)*. Saarbrücken, 2002, S. 231–234

129 Vossen 1994 VOSSEN, Gottfried: *Datenmodelle, Datenbanksprachen und Datenbank-Management-Systeme*. Addison-Wesley, 1994

130 W3C 1998 W3C: *QL'98 - The Query Languages Workshop*. URL `http://www.w3.org/TandS/QL/QL98/`, 1998

131 Wall et al. 2000 WALL, Larry ; CHRISTIANSEN, Tom ; ORWANT, Jon: *Programming Perl*. 3rd edition. O'Reilly, 2000

132 Wallis und Nelson 2000 WALLIS, Sean ; NELSON, Gerald: Exploiting fuzzy tree fragments in the investigation of parsed corpora. In: *Literary and Linguistic Computing* 15 (2000), Nr. 3, S. 339–361

133 Wallis und Nelson 2001 WALLIS, Sean ; NELSON, Gerald: Knowledge discovery in grammatically analysed corpora. In: *Data Mining and Knowledge Discovery* 5 (2001), Nr. 4, S. 305–335

134 Xue et al. 2002 XUE, Nianwen ; CHIOU, Fu-Dong ; PALMER, Martha: Building a Large-Scale Annotated Chinese Corpus. In: *Proceedings of the 19th International Conference on Computational Linguistics (COLING 2002)*, 2002

135 Yu und Meng 1998 YU, Clement T. ; MENG, Weiyi: *Principles of Database Query Processing for Advanced Applications*. Morgan Kaufmann, 1998