Parsing Modul für ein Touristen-Informationssystem[1]

Thomas Cottmann — Blücherstraße 3 — D 69115 Heidelberg

1. September 2001

[1]Das Modul ist Teil des Leistungsnachweises zu Herrn Prof. Dr. P. Hellwigs Hautpseminar: *Sprachgenerierung*, Lehrstuhl für Computerlinguistik, Universität Heidelberg SS 2001

Zusammenfassung

Dieses Dokument beschreibt ein Parsing Modul für ein mobiles Informationssystem für Touristen. Das System verarbeitet natürlich sprachliche Eingaben zu vollständig generierten natürlich sprachlichen Antworten. Fragen und Antworten fallen in 3 Kategorien:

Routenbeschreibung, Standortbestimmung, Hintergrundinformationen zu einer Sehenswürdigkeit

Routenbeschreibung ist ein Weg vom aktuellen Standort zu einem Ziel, *Standortbestimmung* gibt dem Benutzer des Systems seinen aktuellen Standort und *Hintergrundinformationen zu einer Sehenswürdigkeit* liefert Informationen aus einer Datenbank zu Gebäuden und anderen Objekten, welche sich im Umfeld des Benutzers befinden.

Das Parsing Modul in diesem System klassifiziert die Benutzereingabe in die 3 Kategorien und filtert die für die Anfrage notwendigen Parameter heraus. Das Verfahren orientiert sich an Gregory Grevestettes Aufsatz *Light Parsing as Finite State Filtering* [1]. Es handelt sich dabei um flaches Parsing mit regulären Ausdrücken, welches eine hohe Robustheit des Moduls gewährleistet. Im Modul passiert die Eingabe 4 Filterstufen, die den Informationsgehalt reduzieren bzw. signifikante Informationen (Schlüsselwörter und Phrasen) herausfiltern. Ein in die vierte Stufe integriertes Wertungssystem gewichtet die verbliebenen Informationen, repräsentiert als Tokens, und ordnet die Anfrage einer der 3 Kategorien zu. Die ersten drei Filter basieren auf *Flex,* dem *Fast Lexical Analyzer* Tool, welches mit regulären Ausdrücken arbeitende Transducer generiert. Der vierte Filter enthält ebenfalls einen Automaten, dieser weist den Tokenkombinationen der Eingabe Werte zu, um die Entscheidung über die Kategorie zu treffen. Die Konstruktion des Automaten der letzten Stufe beruht zum größten Teil auf der Analyse eines aus 66 Nutzeranfragen bestehenden Korpus, welches aus den Angaben von 10 Personen zusammengestellt wurde.

Inhaltsverzeichnis

Kapitel 1

Überblick

Das vorliegende Parsing Modul für ein Touristen-Informationssystem klassifiziert natürlich sprachliche Benutzeranfragen in drei Kategorien und ermittelt die zugehörigen Parameter, welche mit der Kategorieinformation an nachgeschaltete Module zur Weiterverarbeitung übergeben werden können. Das Verfahren orientiert sich an Gregory Grevestettes Aufsatz *Light Parsing as Finite State Filtering* [1]. Es handelt sich dabei um flaches Parsing mit regulären Ausdrücken, welches eine hohe Robustheit des Moduls gewährleistet. Der Prozeß kann als Filterungsprozeß angesehen werden, da die Token[1] der Eingabe nach gewissen Regeln reduziert werden — nur die gewünschte Information kann den Filter passieren. Und um beim technischen Begriff des Filters zu bleiben, das vorliegende Parsing Modul benutzt eine ganze Filterbank, um das Endergebnis zu erreichen. Im Prinzip wird die Eingabe schrittweise in eine Tokenkette umgewandelt. Der Prozeß des Filterns wird in Kapitel 2 im Detail vorgestellt. Nachdem die Eingabe den letzten Filter durchlaufen hat, erfolgt die Endauswertung über einen Automaten, der zwei Funktionen erfüllt:

- Klassifikation der (gefilterten) Benutzeranfrage in 3 Kategorien
- Ermittlung eventueller Parameter zur jeweiligen Kategorie

Die Endauswertung folgt auch exakt obiger Reihenfolge, erst die Klassifikation in die drei Kategorien (*Routenbeschreibung, Standortbestimmung, Hintergrundinformationen zu einer Sehenswürdigkeit*), dann die Bestimmung zugehöriger Parameter. Die Klassifikation in die drei Kategorien hat deshalb oberste Priorität, da für zwei der drei Kategorien Parameter nicht zwingend erforderlich sind.

[1] Token sei in diesem Zusammenhang eine Einheit, die ein natürlich sprachliches Wort oder Phrase ist. Token können in Form von Tags, <*NP='das alte Haus'*>, erscheinen.

Sofern *Standortbestimmung* nicht als Anfrage zum Standort eines Objektes verstanden wird, kann ein Parameter entfallen. Bei den *Hintergrundinformationen zu einer Sehenswürdigkeit* kann unter Umständen ebenfalls auf den Parameter Referenzobjekt, also das Objekt zu dem Informationen ermittelt werden sollen, entfallen, wenn dieser von einem anderen Modul des Systems ebenfalls ermittelt werden kann. So ist es sicherlich praktikabel als Objekt jeweils diejenige Sehenswürdigkeit zu wählen, die dem Standort des Benutzers am nächsten ist, dieser ist dem System aber immer 'bekannt'. Zu diesen Überlegungen kommt noch ein weiterer Aspekt, der in der Forderung nach Robustheit und Natürlichkeit des Systems begründet ist. Es erscheint durchaus sinnvoll jede Anfrage zu klassifizieren, auch wenn Parameter fehlen oder gar die Klassifikation selbst auf unzureichender Analyse beruht. Im letzteren Fall klassifiziert das Modul die Anfrage immer als *Routenbeschreibung*! Dieses Vorgehen stützt sich auf die Annahme, daß dies die vom Benutzer am häufigsten gewünschte Funktion ist. Und obwohl im aktuellen Projektstadium noch kein Dialogmanagement vorgesehen ist, bereitet eine solche Vorgehensweise eine solche Funktionalität vor bzw. sie unterstützt diese. Ein menschlicher Kommunikationspartner tendiert ja ebenfalls dazu, eine nicht verstandene Anfrage eher aus der Kommunikationssituaion heraus zu interferieren, als sofort mit einem "Ich habe sie nicht verstanden!" zu antworten. Es erscheint also aus psychologischen Erwägungen adäquat eine falsche Antwort einer Standardantwort vorzuziehen. Fehlen z.B. die Parameter für eine Route, so ist die Rückfrage des Systems "Habe ich Sie richtig verstanden, daß Sie mich nach einer Wegbeschreibung gefragt haben?" besser, weil sie Aufmerksamkeit und Interesse suggeriert. Ein geeignetes Dialogmanagment kann somit helfen den Parsingaufwand bzw. Mehrfacheingaben durch den Benutzer zu reduzieren.

Die Klassifikation selbst erfolgt mittels eines Automaten (Kapitel 3.3), der dem Vorkommen gewisser Tokenkombinationen eine numerische Gewichtung innerhalb der 3 Kategorien zuweist. Der Aufbau des Automaten beruht auf einer quantitativen Auswertung des Korpus (Kapitel 3.1), welche in ein Punktesystem (Kapitel 3.2) überführt wurde.

Das vorletzte Kapitel (4) stellt die Quellcode-Seite[2] der Software dar, während das letzte Kapitel (5) versucht das Parsing Modul einzustufen und mögliche Weiterentwicklungen aufzuzeigen.

Zusätzlich gibt es einen ausführlichen Anhang, der Teilergebnisse des Filterungs-Prozesses und die Quellcodes der einzelnen Filterstufen enthält.

[2] Zusätzlich ist dieser Dokumentation eine CD beigefügt, die das Korpus und alle Programmteile im Quellcode enthält.

Kapitel 2

Parsen als Filterungsprozeß

2.1 Das Prinzip

Wie schon eingangs erwähnt, wird in diesem Modul eine Filterbank benutzt, um die gewünschten aus Informationen herauszufiltern. Parsing heißt bei dieser Methode also verschiedene Filter hintereinander. Dabei wird immer die gesamte Eingabe Nominalphrasen, wobei nicht erkannte/unbearbeitete Elemente zunächst im Kontext verbleiben. Einer der Hauptvorteile dieser Methode ist, daß der Informationsgehalt bzw. die Art der Token am Ausgang jeder Filterstufe bzw. am Ende des Prozesses überschaubar bzw. vorhersagbar ist. Dazu ein einfaches Beispiel:

Der Baum steht vor dem Haus, das schon hundert Jahre alt ist.

In diesem einfachen Satz sollen alle Nominalphrasen erkannt werden. Eine einfache aber effektive Strategie zu diesem Problem ist es, sich alle Phrasen anzusehen, die durch einen Artikel oder Demonstrativpronomen werden. Dieser Ansatz vermeidet die Nutzung eines Lexikons, welches unter Umständen nicht alle Nomen enthält. Ein solcher 'Artikelfilter' würde folgendes Ergebnis liefern:

<DET> Baum steht vor <DET> Haus,<DET> schon hundert Jahre alt ist.

Auf diesen ersten Filter folgt nun ein zweiter, der eine Nominalphrase wie folgt definiert:

Jedes Wort, welches mit einem Großbuchstaben beginnt und direkt auf einen *<DET>* Token folgt, ist eine *<NP>*.

<NP='<DET> Baum'> steht vor <NP=' <DET> Haus'>,<DET> schon hundert Jahre alt ist.

Mit diesen 2 Filtern sind schon 2 von 3 Nominalphrasen erkannt. Zugegeben, man hätte diese Aufgabe auch mit einem einzigen Filter bzw. regulären Ausdrücken erledigen können; die Vorteile dieses Vorgehens zeigen sich aber bei der Fortentwicklung des Systems und bei der Fehleranalyse. So tritt bei Parsern, die auf Regelgrammatiken basieren, immer wieder das Problem auf, daß das Systemverhaltenunerwünschte Nebeneffekte aufzeigt, nachdem ein neues sprachliches Phänomen bzw. eine neue Regel integriert wurde. Das Konzept der kaskadierten Filter hilft hier die Komplexität zu reduzieren und Fehlerquellen im System einfacher zu lokalisieren.

Soll z.B. das Phänomen, daß Zahlwörter und Zahlen ebenfalls vor Nomina stehen, aufgenommen werden, so reicht es einzig den ersten Filter zu erweitern, der zweite Filter kann dabei völlig unangetastet bleiben.

<DET> Baum steht vor <DET> Haus,<DET> schon <DET> Jahre alt ist.

<NP='<DET> Baum'> steht vor <NP=' <DET> Haus'>,<DET> schon <NP='<DET> Jahre'> alt ist.

Eine Veränderung an beiden Stufen ist vorzunehmen, wenn Nominalphrasen aus Artikeln und Adjektiven erkannt werden sollen.

Das ist der alte Baum.

<DET> ist <DET> <ADJ> Baum.

<DET> ist <NP='<DET> <ADJ> Baum'>.

An diesem Beispiel läßt sich eine andere Interpretation der Filterstufen aufzeigen. Die erste Stufe transformiert nur einzelne Worte (Lexikonphase), die zweite faßt Token und Wörter zu neuen Einheiten zusammen. Eine dritte Stufe wäre denkbar, die alle verblieben Wörter entfernt, damit eine vierte Stufe auf reiner Tokenebene wiederum neue Einheiten schafft.

An dieser Stelle soll auf einen weiteren Vorteil dieses Verfahrens hingewiesen werden. Obwohl die Nomina dem System nicht bekannt sind, z.B. nicht im Lexikon stehen, existiert dennoch eine Nominalphrase, die Nomina selbst (oder andere Bestandteile) sind in ihr gekapselt. Aufgrund dieser Kaspelung ist es möglich auf der nächst höheren Stufe mit Nominalphrasen zu arbeiten, selbst wenn diese im Detail unbestimmt sein sollten. So wäre es ebenfalls denkbar eine Nominalphrase noch einfacher zu definieren: Ein Artikelwort, welchem unter Umständen eine beliebige Anzahl von kleingeschriebenen Wörtern auf jeden Fall aber ein Nomen folgt.

Das alte rote verrostete Auto stand auf dem staubtrockenen hinteren Parkplatz.

Ein solcher Satz enthielte trotz des eher seltenen Adjektivs *staubtrocken* zwei gültige Nominaphrasen, die weiterverarbeitet werden könnten.

<NP='<DET> alte rote verrostete Auto'> stand auf <NP='<DET> staubtrockenen hinteren Parkplatz'>.

Zum Abschluß ein letztes Beispiel, das eine Vorstellung von der Entwicklungsarbeit bei Nutzung der Filtertechnik geben soll.

Das ist der 100 Jahre alte Baum.

Nehmen wir an, es kommt wieder die ursprüngliche Filterstufe zum Einsatz, die Artikel, Demonstrativpronomina, Zahlen, Zahlwörter und Adjektive erkennen kann. Nimmt man keine Veränderungen an der zweiten Stufe vor, sieht der Entwickler sich mit folgendem Ergebnis konfrontiert:

<DET> ist <DET> <DET> Jahre <ADJ> Baum.

<DET> ist <DET><NP=' <DET> Jahre'> <ADJ> Baum.

Das Ergebnis wäre das gleiche Ergebnis ergibt sich, wenn man Zahlen und Zahlwörter als Adjektive klassifiziert. Was ist also zu tun? Welche Strategien könnten verfolgt werden? In 'traditioneller Denkweise' könnte man die Definition der Nominalphrase komplexer gestalten. Dies kann aber zu Lasten der Robustheit gehen — *100 Jahre* als Nominalphrase zu klassifizieren ist ja prinzipiell auch nicht falsch.

100 Jahre ist eine lange Zeit.

Einfacher und im Stil des Verfahrens wäre es daher dieses Problem auf einer nächsten Stufe zu lösen. Auf dieser Ebene sei eine Nominalphase wie folgt definiert:

Eine <NP> besteht aus einem <DET> + <ADJ> + Nomina

In einem fiktiven System mit Lexikonphase ergeben sich mit dieser Definition folgende Ergebnisse:

1. *Das ist der 5 Hundejahre alte Rascal.*
2. *Das <VERB INF='sein' VALUE='3rdSg'> der 5 Hundejahre alte <LEX='Rascal' TYPE='Eigenname'>*
3. *<DET> <VERB INF='sein' VALUE='3rdSg'> <DET> <DET> Hundejahre <ADJ> <LEX='Rascal' TYPE='Eigenname'>*
4. *<DET> <VERB INF='sein' VALUE='3rdSg'> <DET> <NP='<DET> Hundejahre '><ADJ> <LEX='Rascal' TYPE='Eigenname'>*
5. *<DET> <VERB INF='sein' VALUE='3rdSg'> <NP='<DET> <NP='<DET> Hundejahre '><ADJ> <LEX='Rascal' TYPE='Eigenname'>'>*

6. *<SENTENCE='<DET> <VERB INF='sein' VALUE='3rdSg'> <NP='<DET> <NP='<DET> Hundejahre '><ADJ> <LEX='Rascal' TYPE='Eigenname'>'>' TYPE='This is a house.' >*

In diesem Beispiel wurde angenommen, daß die erste Stufe Zugriff auf ein einfaches Wortformenlexikon für Verben und Nomina hat. Für die Analyse kommen hier 5 Filterstufen zum Einsatz. Der Satz wurde vollständig geparst, obwohl das Wort *Hundejahre* nicht im Lexikon eingetragen ist. Durch die Kapselung des unbekannten Wortes wurde erreicht, daß die vierte Stufe die gesamte Nominalphrase erfassen kann bzw. die fünfte Stufe diese, unabhängig von ihrem Inhalt, benutzen bzw. weiterverarbeiten kann.

2.2 Anwendungsgebiete

Parsen als Filterungsprozeß ist ein sehr robustes Verfahren, welches auch bei großen Textmengen und komplexen Sätzen immer Ergebnisse liefert. Es eignet sich besonders für Parsinganwendungen, die dem *Information Retrieval* nahe stehen. So lassen sich Nominalphrasen und grobe syntaktische Strukturen schnell und zuverlässig erkennen, selbst wenn kein großes Lexikon oder morphologische Analyse zur Verfügung steht. Besonders in der Interaktion mit einem menschlichen Benutzer wiegen die systemimmanenten Nachteile nicht so schwer oder können sogar zur Stärke werden. Eine Suchanfrage im Internet wie,

Ich hätte gerne alle Zeitungsartikel zum UN Gipfel in Genf vor 2 Jahren, aber nur die von deutschen Zeitungen?

, könnte sich bei der Filterung wie folgt darstellen:

1. *Ich hätte gerne alle Zeitungsartikel zum UN Gipfel in Genf vor 2 Jahren, aber nur die von deutschen Zeitungen?*

2. *<I> <NICE2HAVE> <ALL> <KEYWD='Zeitungsartikel'> zum <KEYWD='United Nations'> Gipfel in <KEYWD='Genf '> <BEFORE> <#2> <YEARS> <PUNCT><BUT> <ONLY> <DET> von <KEYWD='deutsch'> <KEYWD='Zeitung'>?*

3. *<I> <NICE2HAVE> <ALL> <KEYWD='Zeitungsartikel'> zum <KEYWD='United Nations'> Gipfel in <KEYWD='Genf '> <TIME:='01.01.1998'>'><PUNCT><BUT> <ONLY><NP='<DET> von <KEYWD='deutsch'> <KEYWD='Zeitung'>'>?*

4. *<I><NICE2HAVE><ALL><KEYWD='Zeitungsartikel'>zum<KEYWD='United Nations'> Gipfel <KEYWD='Genf '> in <TIME:='01.01.1998'>'><PUNCT><BUT> <ONLY><NP='<DET> von <KEYWD='deutsch'><KEYWD='Zeitung'>'>?*

5. *<SEARCH:<ALL><KEYWD='Zeitungsartikel'><KEYWD='United Nations'><STRING='Gipfel'><KEYWD='Genf'><TIME:='01.01.1998'>'><CONSTRAIN TYPE='log.and'><ORIGIN><NP='<KEYWD='deutsch'><KEYWD='Zeitung'>'>>*

Diese durchaus realistische Umformung hat die Eingabe als Suchanfrage kategorisiert, welche aus zwei Teilen besteht:

1. Die Suchanfrage
2. Eine einschränkenden Bedingung.

Beide Teile sollten von einer Suchmaschine bearbeitet und zu einer zielgerichteten Suchanfrage zusammengestellt werden können. Einzig die Zeitangabe bedarf eventuell einer Rückfrage an den Nutzer; eine Antwort des System könnte sich dann wie folgt darstellen:

Suche nach folgenden Schlüsselwörtern	*United Nations, Genf*
untergeordnete Schlüsselwörter	*Gipfel*
Sprache der Dokumente	*deutsch*
Type der Dokumente	*Presse und Nachrichten*
Erstellungszeitraum der Dokumente	*01.01.1998 - heute*

Eine Suchmaschine mit solcher Funktionalität, erspart dem Nutzer sich von Anfang an mit komplexen Eingabefeldern oder Formularen beschäftigen zu müssen. Da es sich bei Suchanfragen um eine überschaubare Menge von syntaktischen Strukturen handelt, scheint es sinnvoll hier ein flaches Parsingverfahren, wie es das Filterungsverfahren ist, einzusetzen. Für das Anwendungsgebiet eines Touristen-Informationssystems gilt ähnliches. Am Anfang der Konstruktion eines Filtersystems steht immer eine gründliche Analyse der Eingabedaten, um die ersten Filterstufen zu 'justieren'.

Kapitel 3

Vom Token zur Entscheidung

3.1 Die Analyse des Korpus

Jeder technische Filter ist an sein Eingangssignal angepaßt, dies gilt im übertragenen Sinne auch für dieses System. Da von Filterstufe zu Filterstufe das Originalsignal, die Benutzeranfrage, modifiziert wird, sind gerade die ersten Filter besonders wichtig. Was hier an Information verloren geht, steht allen weiteren Stufen nicht mehr zur Verfügung. Es sind die ersten Filter, die den Eingabestring in eine Kette von definierten Token umwandeln. Dieser Prozeß bedarf einer genauen Planung, die auf einer genauen Analyse der Eingabedaten basieren muß.

3.1.1 Schlüsselwörter und häufige Konstruktionen

Im verwendeten Korpus (siehe Anhang A) sind es mehr Vokabeln als syntaktischen Konstruktionen,welche variieren. Dies liegt hauptsächlich an der Tatsache, daß die sehr freie Satzstellung des Deutschen in Fragesätzen nicht zum Tragen kommt bzw. sie dort sehr eingeschränkt ist. So sind 31 der 66 Anfragen Fragen, die mit einem Fragewort beginnen. Diese Fragewörter und Fragekonstruktionen wie "Ist es ...?", werden einheitlich zum <*QUESTION*> Token zusammengefaßt. Fragen, die *Ich* als Subjekt haben, sind fast ausschließlich im Bereich der *Routenanfrage* zu verzeichnen; daher gibt es hierfür auch einen Token: <-*I*-> Es sind insgesamt 10 verschiedene Token, die bei der Filterung in der ersten Stufe[1] gesetzt werden.

[1]Die Eingabe ist an dieser Stelle nur vereinfacht dargestellt! Eine genauere Beschreibung der möglichen Eingabe bzw. der regulären Ausdrücke findet sich in Kapitel, 4 und im Anhang.

Eingabe	Token
{lexikalisches Wort}	<LEX='*Lexikoneintrag*'>
ich, mich	<-I->
was, wie, 'ist ...'	<QUESTION>
kommen, suchen	<GET-TO>
Weg, Route	<PATH>
schnell, schnellste, kürzeste, 'am schnellsten'	<SPEED>
Nähe, hier	<VICINITY>
nächst	<NEXT>
Standort, Hilfe, 'Wo ... ich?', 'Wo ... ich hier?'	<LOCATION>
Information, Info, Name, heißt, wissen, bauen, *Architket, Gründer, Wann, Wer, 'Ist dies ...', 'Ist das ...'*	<INFORMATION>

Abgesehen von den Token *<QUESTION>*, *<LOCATION>* und *<INFORMATION>*, welche die Repräsentation von einfachen Phrasen sein können, werden in der ersten Stufe nur signifikante Wörter herausgefiltert. Die nächste Stufe filtert Nominalphrasen und Präpositionlaphrasen, die mit der Präposition *zum* eingeleitet werden. Zu den obigen 10 Token kommen somit zwei weitere hinzu:

<NP='*Nominalphrase*'>   <DESTINATION='*Präpositionalphrase*'>

Nominalphrasen sind in dieser Umgebung Wörter oder Phrasen, die mit einem Artikel oder Demonstrativpronomen eingeleitet werden. Für die Zielangabe werden zunächst alle Wörter, die auf die Präposition *zum* folgen, als mögliche Parameter betrachtet. Die Analyse des Korpus läßt aber für unseren Anwendungsfall folgende Vereinfachung zu:

Sofern Höflichkeitspartikel fehlen, oder die Verbform nicht komplex (= zweigeteilt wie z.B. in Abfrage Nr. 26) ist, befindet sich die gesuchte Information, der Hauptparameter der Anfrage, in der Regel direkt zwischen der Präposition und dem Satzendezeichen. Diese Strategie schlägt aber fehl, wenn die Suchanfrage ein Objekt ist, welches mehrteilig ist.

Ich möchte zum ***Bahnhof****.* versus *Ich möchte zum* ***Zentralen Betriebshof der HSB****.*

Letzterer Fall ist allerdings unproblematisch, wenn der ganze Begriff oder Bestandteile im Lexikon stehen. In diesem Fall kann zunächst den schon vorher ermittelten Teilen/Token der Vorzug als potentielle Parameter gegeben werden.

Ich möchte zum UGC Kino. → *Ich möchte zum UGC <LEX='CINEMA'> .*

In jedem Fall ist es sicherlich von Vorteil der nächsten Komponente nur solche (Such-)parameter zu übermitteln, welche auch als bestimmt bzw. dem Gesamtsystem bekannt gelten; obwohl es prinzipiell, wie hier angedeutet, möglich ist, Parameter nur aufgrund der syntaktischen Verhältnisse zu ermitteln.

Nachdem die ersten beiden Filter die Eingaben in Kombinationen von 12 Token umgewandelt, entfernt ein dritter Filter die verbliebenen unbearbeiteten Teile. Die natürlich sprachliche Eingabe ist wird jetzt einzig und allein durch eine Kombination von Tags[2] repräsentiert. Im nächsten Schritt der Korpusanalyse gilt es nun diese Muster in die drei Kategorien *Routenbeschreibung, Standortbestimmung* und *Hintergrundinformationen zu einer Sehenswürdigkeit* zu klassifizieren. Hierzu erscheint eine graphische Darstellung der quantitativen Verteilung aller Tags auf die drei Kategorien sinnvoll.

3.1.2 Verteilung und Klassifikation der Tags

Ausgehend von der 'manuellen' Unterteilung des Korpus in die drei Gruppen (1–39 *Routenbeschreibung*, 40–47 *Standortbestimmung*, 48-66 *Hintergrundinformationen zu einer Sehenswürdigkeit*), lassen sich in jeder Kategorie für jeden Tag seine Häufigkeit bestimmen. Die Ergebnisse dieser Auszählung sind in den Abbildungen 3.1, 3.2 und 3.3 dargestellt.

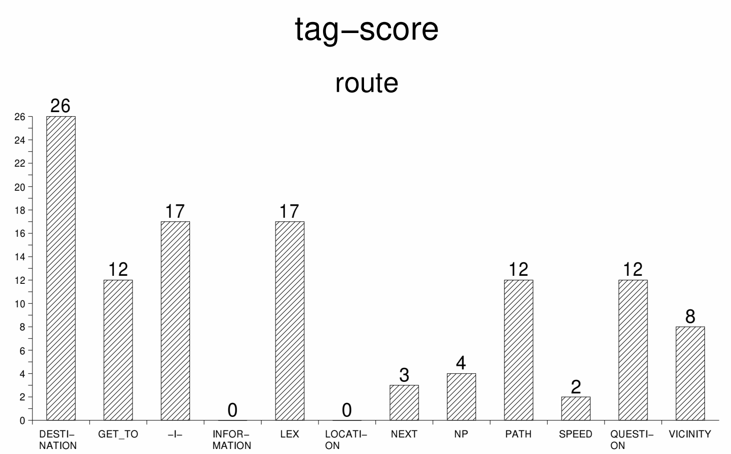

Abbildung 3.1: absolute tag-score for routing queries

Diese absoluten Zahlen können bei der Analyse einer einzelnen Kategorie schon recht hilfreich sein. So fällt auf, daß es in jeder Gruppe einen Tag gibt, welcher am häufigsten vorkommt und dabei eng mit der Kategorie verknüpft ist. Es handelt sich dabei um folgende Tags: *<DESTINATION>* *<LOCATION>* und

[2] Ich möchte an dieser Stelle vom Begriff des Tokens zu dem des Tags übergehen, da ab dieser Stufe der Ursprung der Tags keine direkte Relevanz für die weitere Verarbeitung hat.

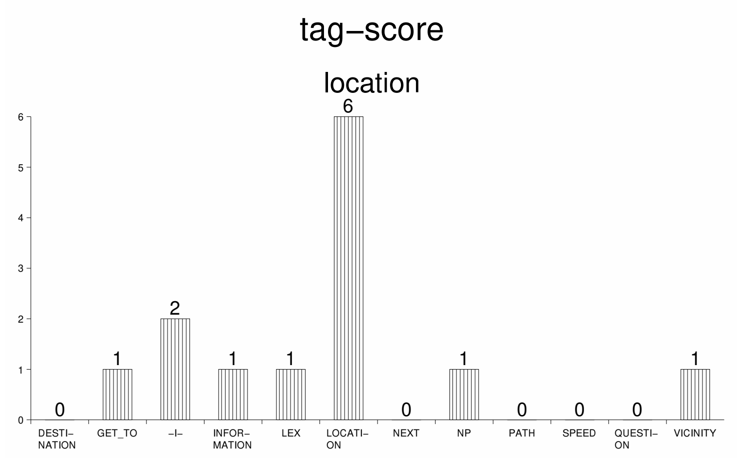

Abbildung 3.2: absolute tag-score for location queries

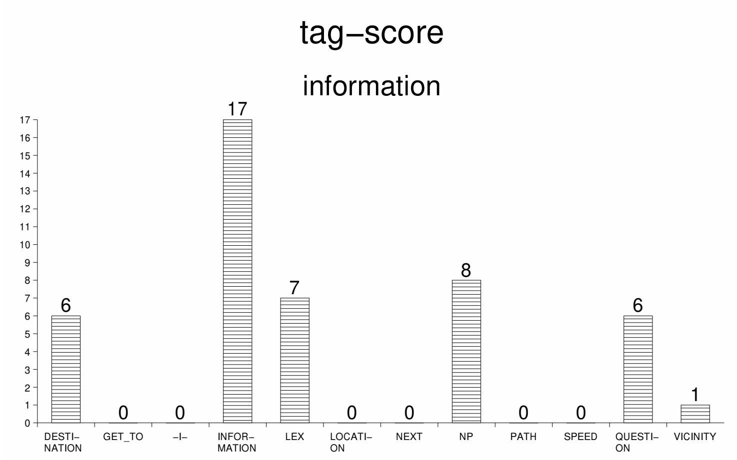

Abbildung 3.3: absolute tag-score for information queries

<INFORMATION> Die Hauptmerkmale der ersten Filterstufen lassen sich mit diesen Daten schon einstellen und beurteilen.

Für eine Gesamtanalyse, also über die die Einzelkategorien hinweg, müssen obige Daten aber noch in Verhältnis zur Datenbasis gesetzt werden, da die Anzahl der Anfragen pro Gruppe nicht identisch ist. Die Abbildungen 3.4, 3.5 und 3.6 zeigen die relative Häufigkeit aller Tags innerhalb der Kategorien.

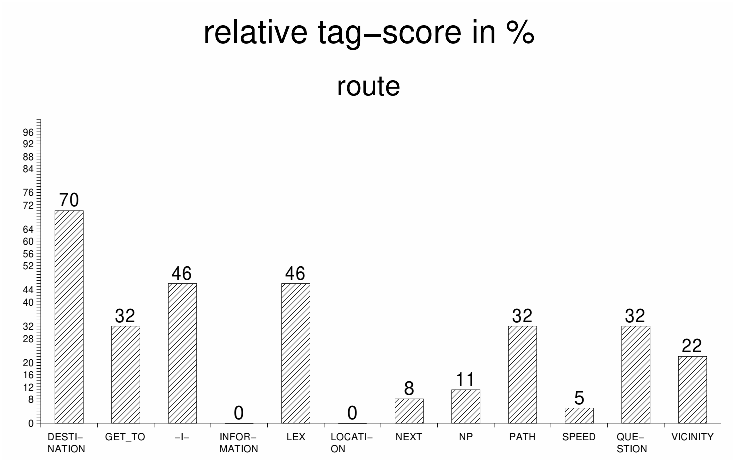

Abbildung 3.4: relative tag-score for routing queries

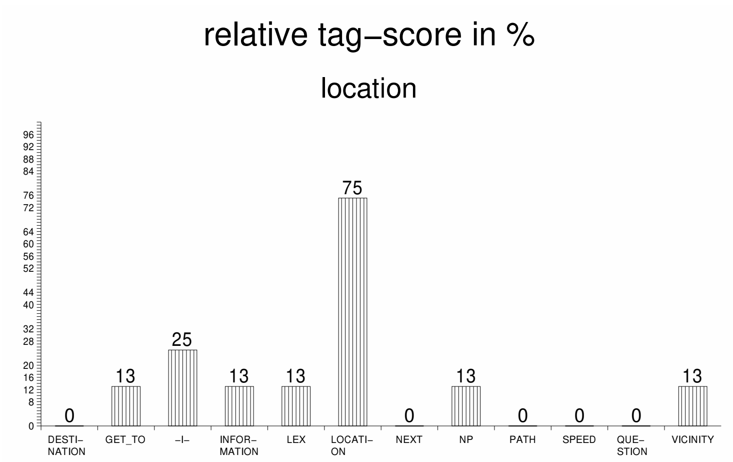

Abbildung 3.5: relative tag-score for location queries

Diese relativen Zahlen können zur besseren Übersicht in einem einzelnen Diagramm zusammengefaßt werden (Abbildung 3.7).

Eine alternative Darstellung zeigt Abbildung 3.8. Die Hauptentwicklungsarbeit im Bereich der höheren Filterstufen stützt sich hauptsächlich auf diese Darstel-

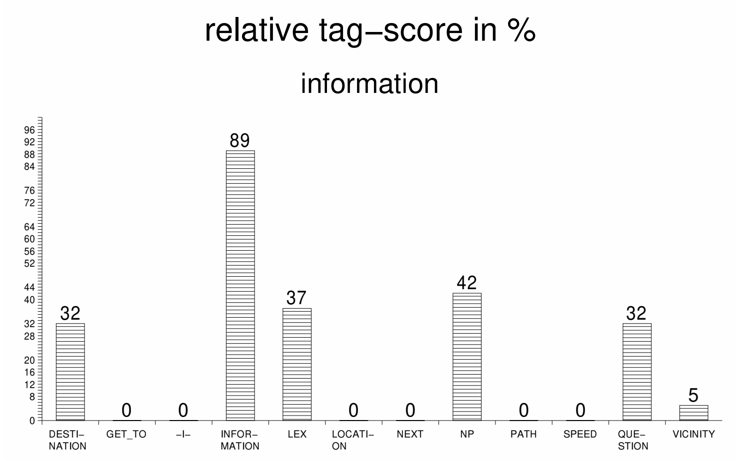

Abbildung 3.6: relative tag-score for information queries

Abbildung 3.7: overview of all relative tag-scores

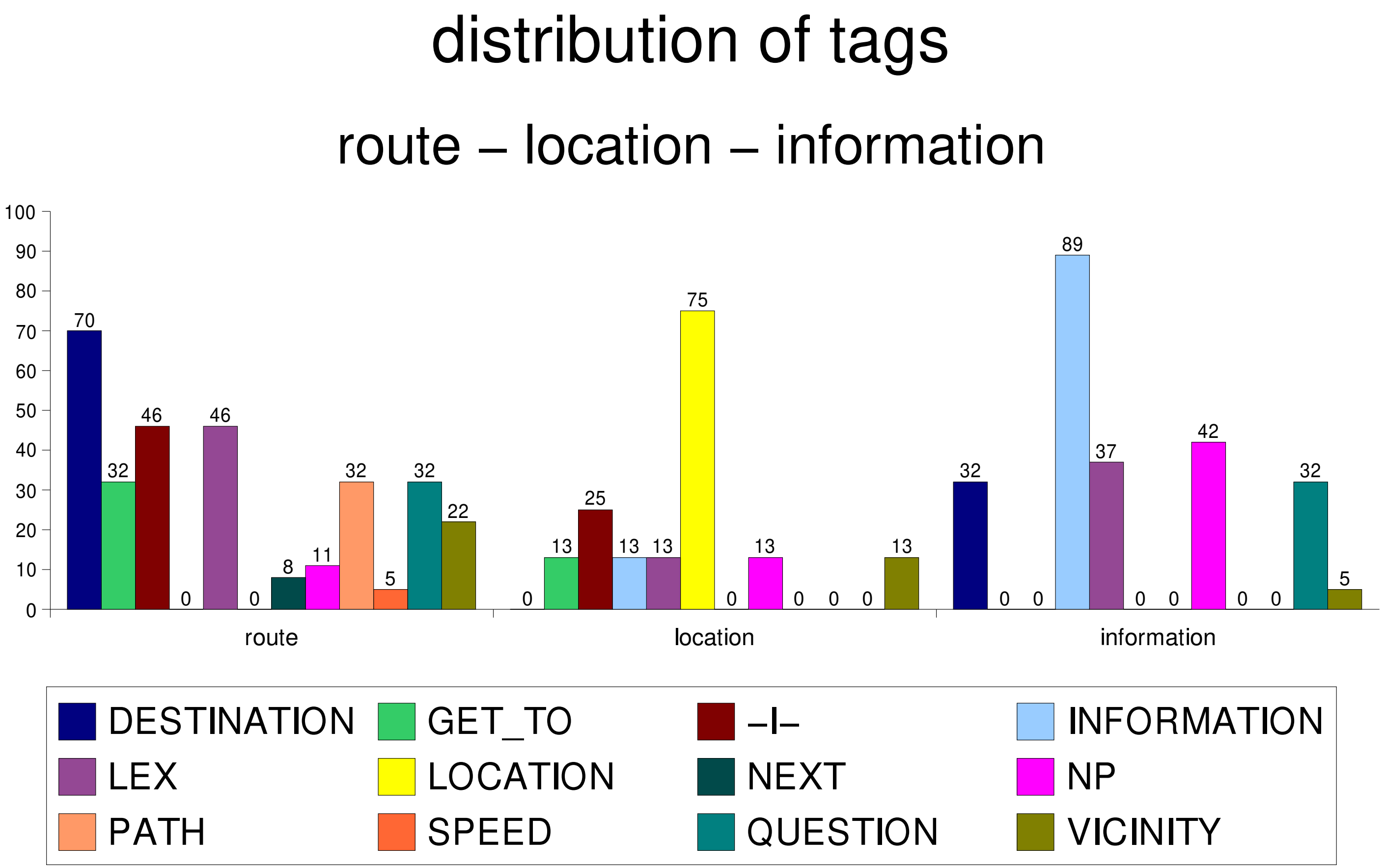

Abbildung 3.8: distribution of tags

lung . Aus ihr wurde das Punktesystem (Kapitel 3.2) abgeleitet und der finale Transducer (Kapitel 3.3) entwickelt. Hauptaugenmerk ist hierbei vor allem auf diejenigen Säulen/Tags zu richten, die in einer oder mehreren Kategorien den Wert Null aufweisen bzw. nicht vorhanden sind. Im Vergleich mit den übrigen Kategorien lassen sich dann eventuell eindeutige Indikatoren finden. So tritt z.B. der *<LOCATION>* Tag nur in der Kategorie *Standortbestimmung* auf. Das Gleiche gilt für den *<NEXT>* Tag, er ist im Korpus nur unter den *Routenbeschreibungen* zu finden. Allerdings ist das Vorkommen dieses Tags innerhalb des Korpus recht gering, selbst wenn es ein gutes Entscheidungskriterium ist.

Abgesehen von diesen recht simplen Kriterien, sind es jedoch die Kombinationen von verschiedenen Tags, welche Analyse der Anfrage ermöglichen. Auch hier stellen die Diagramme eine wichtige Hilfe bei der Auswertung dar. Abbildung 3.4 zeigt deutlich, daß das die Tags *<DESTINATION>* und *<-I->* als potentielle Kombination in Routenanfragen in Frage kommen können, da sie am häufigsten vorkommen. Tatsächlich kommt dieses Paar in 11 der 39 Anfragen vor — wichtiger aber ist, daß dieses Paar in keiner der anderen Kategorien vorkommt. Ähnliches gilt für dieKombination von *<DESTINATION>* mit *<LEX>* oder *<GET_TO>*. Anfrage Nr. 8 des Korpus (*Wie komme ich von hier zur Alten Brücke?*) ist ein Beispiel in dem sogar alle 4 dieser Tags vorkommen! Ein einfaches System diese Kombinatorik zu verarbeiten ist ein Punktesystem.

3.2 Ein Punktesystem

Um die Benutzeranfragen in die drei Kategorien zu sortieren, wird ein einfaches Punkte- bzw. Wertungssystem eingesetzt, welches die Form eines Transducers hat. Dabei wird jedem akzeptierten Eingabesymbol, Tag, ein Ausgabewert zugeordnet, der seinerseits mit einer der drei Kategorien verknüpft ist. Abbildung 3.9 zeigt einen solchen sehr einfachen Transducer, der drei verschiedene Tags akzeptiert.

Unter dem Graph steht das Eingabesymbol, über ihm die Ausgabe bzw. die Aktion. Innerhalb des Wertungssystems sind drei Variablen definiert, welche die drei Kategorien repräsentieren. Diejenige Variable, die am Ende der Analyse den höchsten Wert enthält, bestimmt die Entscheidung. Zur Verfeinerung und Absicherung kann bei einem solchen System ein Schwellwert bzw. Mindestwert[3] implementiert werden, d.h. es müssen bestimmte 'Mindestumsätze'

[3] In der aktuellen Version gibt es derartige Kontrollmechanismus nicht, da es im Gesamtsystem keine Mechanismen gibt auf einen solchen Ausnahmefall effektiv zu reagieren. Es kann aber davon ausgegangen werde, daß ein Wert von 10 ein sehr unsichereres Ergebnis darstellt. Die aktuelle Version klassifiziert 'im Zweifel' jede Anfrage als Routenanfrage.

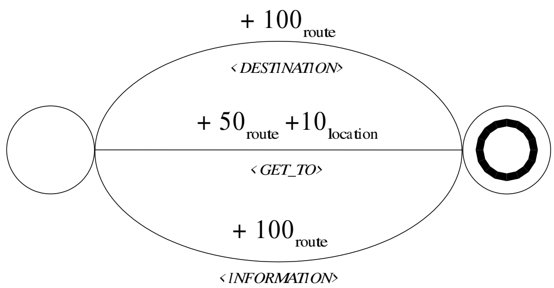

Abbildung 3.9: Punktesystem mit einem einfachen Transducer

und 'Mindestabstände' zwischen den Werten existieren, bevor eine finale Entscheidung getroffen werden kann. Ansonsten wird keine Entscheidung getroffen und eine Hermeneutik setzt ein. Die Werte, welche summiert werden, stellen — neben der Konstruktion der Graphen — die wichtigste Möglichkeit dar die Analyseeinheit zu justieren. Dabei können die Werte in einem laufenden System korrigiert bzw. immer wieder an aktuelle Analysen der Benutzereingaben angepaßt werden, ohne die Grundkonstruktion des Automaten ändern zu müssen. In der Praxis besteht ein solches System aus einer Reihe von Automaten, die kaskadiert oder parallelisiert werden. Dieses Konstruktionsprinzip wirkt sich, wie jede andere Modulbauweise, positiv auf den Entwicklungsprozeß aus. Das Punkteystem macht das Gesamtsystem in weiten Grenzen sehr variabel, ohne dabei seine Stabilität zu gefährden.

3.3 Der finale Transducer

Der finale Transducer (siehe Anhang D) zur Kategorisierung der gefilterten Benutzeranfrage hat 10 Startzustände; sie sind mit römischen Zahlen gekennzeichnet (Abbildung 3.10). Es wird jeweils zum nächsten Startzustand gewechselt, wenn der vorherige Durchlauf nicht zum Endzustand geführt hat. Dies bedeutet, daß für die Entscheidung nur ein Weg, der erste erfolgreiche, aus allen möglichen Wegen benutzt wird; daher ist die Sortierung der Startzustände von hoher Bedeutung. Als Ausgabewerte wurden die Bezeichnungen des Programms benutzt:

TOP HEAVY LIGHT STANDARD

Dabei handelt es sich um symbolische Konstanten. Sie ermöglichen es den Transducer in ein größeres Wertungssystem einzubetten und dabei die tatsächlichen Werte an die Umgebungswerte anzugleichen[4]. TOP repräsentiert den höchsten

[4]Dies ist z.B. wichtig für ein Wertungssystem, das mit Schwellwerten arbeitet. So lassen

Wert, STANDARD den niedrigsten. Ergebnisse, die mit TOP oder höher bewertet wurden, können als eindeutig angesehen werden, während STANDARD Ergebnisse nur wenig Aussagekraft haben.

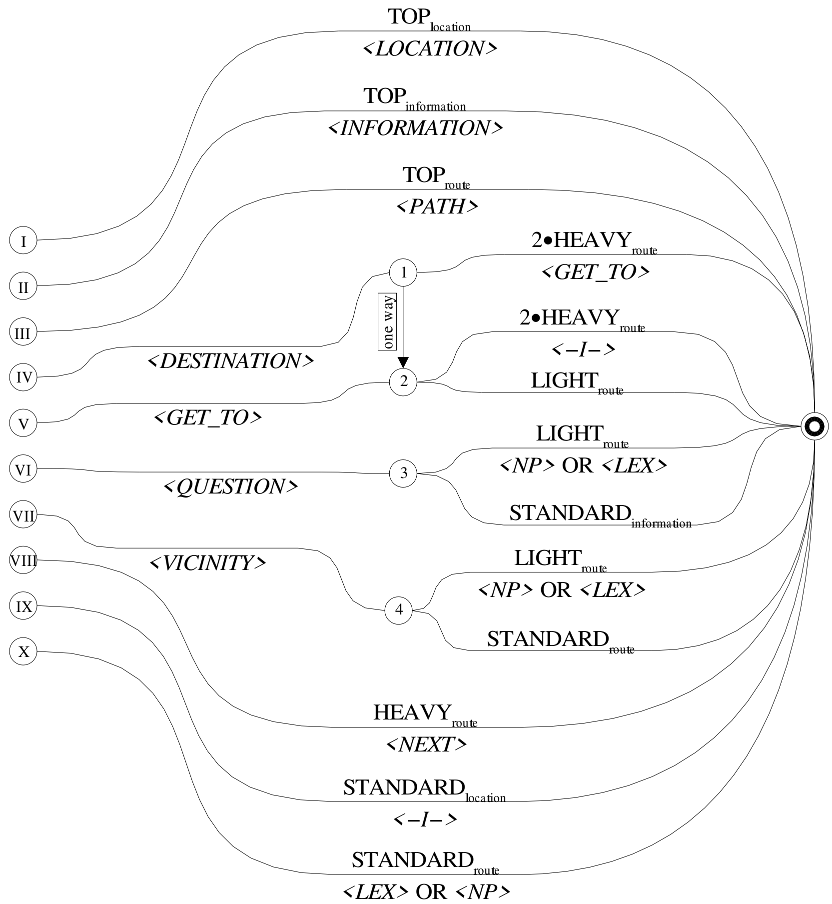

Abbildung 3.10: Transducer zur Kategorisierung der gefilterten Benutzeranfrage

Im Automaten bzw. der Abbildung 3.10 gibt es Abweichungen von der Standarddarstellung: Von Zustand 1 zu Zustand 2 gibt es eine Art 'Einbahnstraße'. Es kann also nur von 1 zu 2, nicht aber von Zustand 2 in Zustand 1 übergegangen werden. Generell werden zunächst alle Pfade verfolgt, die ein Eingabesymbol beinhalten, ein leerer Pfad ist in diesem System immer die letzte Möglichkeit. Zusätzlich sei darauf hingewiesen, daß als Eingabesymbol nur das Vorkommen des Tags in der Anfrage, nicht aber die Reihenfolge der Tags in der Eingabekette bewertet wird. Das Symbol unter dem Graph meint also: *Symbol ist in der Eingabe enthalten.* Sind zwei Symbole durch OR verbunden, so handelt es sich um den gleichnamigen logischen Operator: *Das eine Symbol, das andere Symbol*

sich verschiedene Transducer auf den gleichen Schwellwert trimmen.

oder beide sind in der Eingabe enthalten.

Obwohl der hier dargestellte Transducer recht überschaubar ist, klassifiziert er 63 der 66 Anfragen des Korpus in die richtigen Kategorien. Um die notwendigen Parameter für zwei der drei Kategorien zu bestimmen, ließe sich ein ähnlicher Automat konstruieren; in der vorliegenden Version erfolgt diese Analyse jedoch über einige wenige geschachtelte *if–then–else* Konstruktionen[5] (siehe Anhang D).

[5] Voraussetzung für eine hochwertigerere Analyse ist ein größeres Korpus bzw. die Anbindung an ein (Wortformen-)Lexikon.

Kapitel 4

Die Software

Die Software besteht aus vier Programmteilen[1], welche die Anfragen seriell bearbeiten. Die ersten drei Filter (siehe Anhang C) basieren auf *Flex,* dem *Fast Lexical Analyzer* Tool, welches mit regulären Ausdrücken arbeitende Transducer generiert. Der vierte Filter, die Entscheidungseinheit (siehe Anhang D), enthält ebenfalls einen Automaten, dieser weist den Tokenkombinationen der Eingabe Werte zu, um die Entscheidung über die Kategorie zu treffen.

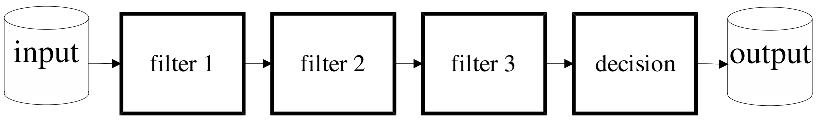

Abbildung 4.1: Aufbau der Software

Der Datenaustausch zwischen den einzelnen Modulen ist textbasiert bzw. findet in Form von Textdateien statt. Um die Entwicklung zu vereinfachen wurde diese Möglichkeit vorgesehen. So ist es möglich statt nur einer Anfrage immer ganze Korpora zu verarbeiten. Der finale Transducer, das Entscheidungsmodul, ist in der Lage Textdateien zeilenweise einzulesen, wobei jede Zeile als gefilterte Anfrage interpretiert wird. Die Benutzeranfrage kann aber auch am Prompt der Shell eingegeben werden.

4.1 Die Filter

Da alle Filter in LEX-Code geschrieben wurde, erfolgt die nachstehende Dokumentation rein auf dieser Ebene. Zur Dokumentation von LEX und dem

[1] Da alle Programmteile in C erstellt sind, ist es kein Problem diese zu einer Einheit zusammenzufügen. Für die Entwicklungsarbeit war diese Trennung aber sehr hilfreich.

generierten C-Code sei auf die ausführliche Dokumentation verwiesen, die diesem Tool beigefügt ist.

4.1.1 Filter #1[2]

Dieser Filter operiert auf der Wortebene. Der erste Teil simuliert den Zugriff auf ein Wortformenlexikon. Dort sind unter einem Lemma die verschiedenen Schreibweisen bzw. Flexionen aufgeführt. Es sei betont, daß für diese Anwendung ein Lexikon, das nur Substantive und ihre Deklinationsformen enthält, völlig ausreichend ist; sind es doch Objekte, welche bei dieser Anwendung im Vordergrund stehen. Nach dem gleichen Prinzip ist der nächste Abschnitt aufgebaut. Dort handelt es sich allerdings bei den Lemmata um Begriffe, die direkt mit der Aufgabenstellung verbunden sind. Eine Aufstellung der Schlüsselwörter dieses Filters findet sich in der Tabelle in Kapitel 3.1.1. Außer den Token, die dem Lexikon entnommen wurden, generiert diese Filterstufe 10 verschiedene Token und eine Nachricht, daß diese Filterebene durchlaufen wurde.

4.1.2 Filter #2[3]

Der nächste Filter filtert nach zwei Phänomenen:

- Zielangaben *<DESTINATION>*, die mit *zu* oder *nach* eingeleitet werden
- Nominalphrasen, die mit Artikel oder Demonstrativpronomen eingeleitet werden

Zu diesem Zweck werden zunächst ein Satz eben jener einleitenden Wörter definiert. Im Quellcode werden diese Angaben dann wie folgt weiter verarbeitet: `{ZUM} {printf("%s",yytext);BEGIN(zum);}`

Wird ein Wort aus dem Satz, der als *ZUM* definiert wurde, erkannt, wird das erkannte Wort wieder ausgegeben und die Bearbeitung am Sprungziel *zum* fortgesetzt.

```
zu[ \t]+((eine[mr])|(diese[rm]))[ \t]+ {printf("%s",yytext);BEGIN(zum);}
```

Werden die Phrasen *zu einem*, *zu einer*, *zu dieser* oder *zu diesem* erkannt, wird die erkannte Phrase wieder ausgegeben und die Bearbeitung ebenfalls am Sprungziel *zum* fortgesetzt.

```
<zum>[^"?"\n]+ {printf("<DESTINATION='%s'>",yytext); BEGIN(0);}
```

[2] Der erste Filter ist auf der CD in 2 Versionen abgelegt: parse0.lex — der Standardfilter parse0.lex — Filter liest Eingabe von stdin

[3] Der zweite Filter ist auf der CD unter dem Namen parse1.lex abgelegt.

Am Sprungziel wird der *<DESTINATION>* Tag gesetzt. Es werden alle Zeichen bis zum Satzende herausgefiltert. Satzende ist hierbei definiert als *?* oder Zeilenwechsel (**C**arriage **R**eturn). Die Zeichenkette zwischen *ZUM* und *?* bzw. *CR* wird so als Zielparameter in den Tag aufgenommen.

Der restliche Teil des Codes beschäftigt sich mit der Erkennung von Nominalphrasen. Dabei werden nach Artikeln, Demonstrativpronomina und *<NEXT>* Tags gefiltert, welche Nominalphrasen einleiten, zB. *ein Baum, jenes Auto, dieser Minister*. Relativkonstruktionen wie, *Ist es die, die den jungen Mann geheiratet hat?*, werden ebenfalls berücksichtigt. Die Nominalphrase ohne einleitenden Artikel oder Pronomen ist wie folgt definiert:

```
<np1>[ \t\n]+(({DIGIT}|{NUMBER})+[ \t\n]+)?([a-züöäß]{3,}[ \t\n]+)?
([ \t\n]*[A-ZÜÖÄ][a-züöäß]+("-"[A-ZÜÖÄ][a-züöäß]+)?)+ {printf("<NP='%s'>",yytext);BEGIN(0);}
```

Diese Definition sei anhand von Beispielen erklärt:

Eingabe	Ausgabe
	<MESSAGE='stratum 1 passed'>
das Haus.	das<NP=' Haus'>.
ein altes Haus.	ein<NP=' altes Haus'>.
ein altes haus.	ein altes haus.
ein grosses altes Haus.	ein grosses altes Haus.
der Zentrale Betriebshof.	der<NP=' Zentrale Betriebshof'>.
die Mondschein Lieblingsessen Mitmach Rumba.	die<NP=' Mondschein Lieblingsessen Mitmach Rumba'>.
der Duo-Einkaufsgutschein.	der<NP=' Duo-Einkaufsgutschein'>.
jener neue Duo-Einkaufsgutschein.	jener<NP=' neue Duo-Einkaufsgutschein'>.
zehn Autos.	zehn Autos.
die zehn Autos.	die<NP=' zehn Autos'>.
die 10 Millionen Mark.	die<NP=' 10 Millionen Mark'>.
<NEXT> Kino?	<NEXT><NP=' Kino'>?

Wie an der dritten Eingabe zu sehen ist, benutzt der Filter das Kriterium der Groß- und Kleinschreibung, um zwischen Adjektiven und Substantiven zu unterscheiden. Adjektive sind als kleingeschriebene Worte definiert, die mindestens aus drei Buchstaben bestehen müssen. Unter Umständen kann auf das Kriterium der Groß- und Kleinschreibung verzichtet werden, wenn Adjektive von einem gesonderten Filter zuvor erkannt worden sind.

4.1.3 Filter #3[4]

In dieser Stufe werden die Reste der Eingabe gelöscht. Zusätzlich werden leere Zeilen entfernt. Der Entscheidungseinheit werden somit pro Zeile eine Anfrage zugeführt, die dann nur noch aus Tags besteht.

4.2 Die Entscheidungseinheit[5]

Da der Transducer schon ausführlich in Kapitel 3.3 beschrieben wurde, sollen in diesem Kapitel nur die wichtigsten Funktionen vorgestellt werden.

Wie schon erwähnt, so gibt es in diesem Programm ein paar Funktionen, die zur Entwicklung implementiert wurden.

`int readfile(char *fname,int lines,int chars);`

Diese Funktion liest eine Datei in einen Zeilen-Array ein, wobei `lines` die Anzahl der Zeilen und `chars` die Anzahl der Zeichen pro Zeile definiert.

`void tag_count(char *str);` benutzt `int match(char *input, char *search);`, um die Anzahl der Tags pro Typ in einer Anfrage zu bestimmen. Zum einen wurde diese Funktion schon benutzt, um die Analyse des Korpus zu unterstützen, deshalb gibt es auch eine Bildschirmausgabe, zum anderen wird diese numerische Analyse von dem Tansducer selbst genutzt. Dieser überprüft das einfache Vorkommen eines Tags unter Nutzung dieser Funktion[6].

Der schon vorgestellte Transducer befindet sich in der Funktion `int machines(int item);`. Dieser Funktion wird die Nummer der aktuellen Zeile als `item` übergeben. Der Rückgabewert ist positiv, wenn der Automat im Endzustand verlassen wurde. Die Werte des Punktesystems sind in den folgenden Variablen: `route.rating`, `location.rating` und `information.rating`

Diese werden ihrer Größe nach sortiert. Der größte Wert 'gewinnt'. Sollte es sich bei dem 'Gewinner' um eine Routenanfrage bzw. Informationsanfrage handeln, so werden die Funktionen `void r_param(int item);` bzw. `void i_param(int item);` benutzt, um die notwendigen Parameter für diese Arten der Anfrage zu bestimmen. Hierzu gibt es zwei structs:

`struct route_struct`

[4] Der dritte Filter ist auf der CD unter dem Namen parse2.lex abgelegt.

[5] Die Entscheidungseinheit mit dem finalen Transducer ist auf der CD unter dem Namen decision.c abgelegt.

[6] An dieser Stelle sei noch einmal darauf hingewiesen, daß bedingt durch diese Funktion der Transducer nur das bloße Vorkommen eines bestimmten Tags in der Eingabe überprüft. Die Anzahl und die Reihenfolge ist damit ohne Bedeutung!

```
{
  int rating;
  char dest_obj[51];
  char on_route_obj[51];
  char speed[11];
} route;
struct information_struct
{
  int rating;
  char query_obj[51];
  char query_type[51];
} information;
```

Der erste Wert, `rating`, ist jeweils für das Punktesystem. `dest_obj` ist das Ziel der Routenanfrage. `on_route_obj` ist ein Wegpunkt bzw. Zwischenziel[7]. `speed` ist ein Parameter, der die prinzipielle Berechnung der Route beeinflußt, z.B. *kürzester Weg*. `query_obj` ist das Objekt zu dem Informationen geliefert werden sollen. `query_type` kann die Auswahl der zurückgelieferten Informationen einengen, z.B. *das Alter des Gebäudes*.

Um die Arbeitsweise des gesamten Systems nachvollziehen zu können sei nicht nur auf den Programmcode im Anhang hingewiesen, zusätzlich befinden sich auf der CD auch die Eingabe- bzw. Ausgabedaten aller Stufen.

[7] Dieser Parameter wird in der aktuellen Version nicht ermittelt!

Kapitel 5

Das Parsing Modul

5.1 Grenzen und Schwachpunkte des Systems

Obwohl das vorliegende System die Anfragen aus dem Korpus gut und richtig parst, sollen gewisse Problematiken nicht unerwähnt bleiben.

So wurde das Mehrfache Auftreten von Tags eines Typs bisher nicht berücksichtigt. Ebenso unberücksichtigt bleiben in der aktuellen Version die Reihenfolge der verschiedenen Tags. Das Korpus in seiner aktuellen Form und Größe, welches für die Entwicklungsphase unabdingbar war, ist zu künstlich und zu klein, um eine realistische Abschätzung über die Güte des Parsers zu treffen. Die 10 Personen, die befragt wurden, wurden alle sehr genau und gleich informiert, was das System leisten soll. Ihnen wurde gesagt, daß die Objekte in den Anfragen ohne Bedeutung sind und daß alle Anfragen vollständige Sätze sein sollen. Auch wurde ihnen die drei Analyse-Kategorien genannt. Um ein realistischereres Korpus zu erhalten, müßten die Anfragen unter realen Bedingungen erfaßt werden:

- Der Personenkreis müßte erweitert und an die tatsächlichen Benutzergruppen angepaßt werden, z.B. sprachliche Schwächen von ausländischen Benutzern.
- Das Eingabemedium der Anfragen muß definiert bzw. berücksichtigt werden. So sind Eingabefehler im aktuellen Korpus ja nicht vorhanden. Ob der Parser Kriterien wie Zeichensetzung oder Groß- und Kleinschreibung nutzen kann, hängt ebenfalls von der Benutzerschnittstelle ab.

- Die unnatürliche Verteilung von Nominalphrasen und Wörtern, die im Lexikon stehen, muß durch den Einsatz eines Lexikons, welches an reale Benutzeranfragen angepaßt ist, beseitigt werden, um eine realsitische Auswertung der Leistung des Systems zu ermöglichen.

Doch selbst, wenn ein größeres Korpus zu Verfügung steht, so werden immer prinzipielle Probleme bleiben. Wie soll das System mit Anfragen umgehen, die nur teilweise 'verstanden' wurden? Ist eine unsichere Antwort einer Rückfrage vorzuziehen? Auch ist die Klassifikation einer Anfrage in die drei Kategorien nicht immer unstrittig. Selbst das vorliegende Korpus ist nicht frei von solchen Fällen. Die Anfragen 46 und 66 sind identisch, finden sich aber in zwei verschiedenen Kategorien[1]. Nehmen wir an, der Nutzer hat sich den Weg zur *alten Brücke* beschreiben lassen und ist dort jetzt gerade angekommen bzw. das Gebäude kommt in Sichtweite. Die Frage, *Ist das die alte Brücke?*, meint in diesem Zusammenhang dann, *Ist das mein Ziel?* . Solche Analysen sind aber schwerlich vom Parser allein zu leisten. Aus ähnlichen Gründen wurden auch die Anfragen 16 und 17 des Korpus nicht benutzt, da die Analyse bzw. die Antworten zu vielfältig sind:

Frage: *Gibt es am Schloss einen Bahnhof?*

Mögliche Interpretationen/Antworten:

1. *Ja.*
2. *Ja, sie kommen wie folgt zum Schloss bzw. Bahnhof ...*
3. *Nein.*
4. *Nein, aber den nächsten Bahnhof finden Sie ...*
5. *Ja, es handelt sich aber nur um einen Regionalbahnhof, der ...*

Wie bei jedem Parser wird es also auch bei diesem System immer Eingaben geben, die syntaktisch und semantisch zu komplex sind. Das Filterungsverfahren stellt für einen solchen erweiterten Anforderungskatalog kein probates Mittel mehr dar! Einfachheit, Schnelligkeit und Robustheit wirken sich im Regelfall negativ auf die mögliche Komplexität der zu erfassenden Phänomene aus. Nur ein Feldversuch kann zeigen, ob der vorgestellte Ansatz für die meisten Anfragen ausreichend erscheint.

[1] Die Software klassifiziert beide als Informationsanfrage.

5.2 Ausblick

Innerhalb dieses Verfahrens gibt es allerdings noch ein großes Entwicklungspotential. Ein 'echtes' Korpus würde eine bessere statistische Analyse und damit bessere Justierung der Filter ermöglichen. Vor allem die Erweiterung des finalen Transducers und die Anpassung des Wertungssystems ist dann möglich. Allerdings müßten vorher wohl eine Reihe von neuen Schlüsselwörtern implementiert werden. Zusätzlich könnte durch den Einsatz eines Lexikons und Erweiterung des *<LEX>* Tags, ein Filter für komplexerere Phrasen definiert werden.

<LEX='alte' TYPE='adj' ><LEX='blaue' TYPE='adj' ><LEX='Schiff' TYPE='sbt'>

Mit Wortklassen oder gar Kasusinformationen können komplexe Nominalphrasen beschrieben werden.

Wie schon in Kapitel 2.2 ausgeführt, so eignet sich dieses Verfahren besonders für Anwendungen in denen es Interaktion mit dem Benutzer gibt. Interaktion bzw. Rückfragen ermöglichen es die Komplexität von Problemen zu reduzieren. Aufgrund der bisherigen Erfahrungen steht zu vermuten, daß die Bestimmung der Parameter immer schwieriger als die Klassifizierung der Anfrage die drei Gruppen sein wird. Mit Rückfragen wie, *Sie möchten Informationen haben, wozu bitte?*, *Habe ich Sie richtig verstanden, daß Sie zur Alten Brücke wollen?* oder *Ich habe das Ziel Ihrer Routenanfrage nicht verstanden.* , lassen sich fehlende recht einfach erfragen, ohne den Parser komplexer werden zu lassen. Das Dialogmanagement ist ja getrennt vom Parser; allerdings können je nach Fortschritt des Dialogs verschiedene Parser bzw. verschiedene Filter aktiviert werden.

Zusammenfassend läßt sich sagen, daß eine funktionelle Weiterentwicklung des Systems nur parallel zu umfangreichen Feldversuchen bzw. realen Einsätzen des Systems erfolgen kann, da die Güte eines Parsers hauptsächlich davon abhängt, wie hoch die Analysequote, also die Anzahl der erkannten Phänomene zur Anzahl der tatsächlichen Phänomene[2], ist. Letztere aber sind nur im realen Betrieb zu erfassen.

[2]Ich halte es nicht für erstrebenswert einen Parser für Phänomene zu konstruieren, welche nicht direkt aus der Praxis gewonnen wurden.

Literaturverzeichnis

[1] Grevenstette, Gregory: *Light Parsing as Finite State Filtering* , Rank Xerox Research Centre Europe, Meylan 1996.

Abbildungsverzeichnis

Kapitel 6

Anhang

- Anhang A — Korpus der Nutzeranfragen
- Anhang B — Nutzernafragen nach der Filterung (Stufen 1–3)
- Anhang C — Flex Quellcode (Filter 1–3)
- Anhang D — C Quellcode des finalen Transducers (4. Filter)
- Anhang E — Inhalt der CD

Anhang A

Korpus der Nutzeranfragen

1–39 *Routenbeschreibung* 40–47 *Standortbestimmung* 48-66 *Hintergrundinformationen zu einer Sehenswürdigkeit*

1. `Wie ist der Weg zum Schloss?`
2. `Wohin geht es zum Schlos?`
3. `Wie komme ich zum Schloß?`
4. `Wo ist die Hauptpost?`
5. `Bring mich zum Bahnhof!`
6. `Wie verläuft die Hauptstraße?`
7. `Komme ich von hier zur Brücke?`
8. `Wie komme ich von hier zur Alten Brücke?`
9. `Gibt es einen Weg von hier zur alten Brücke?`
10. `Gibt es in Heidelberg noch eine ältere Brücke?`
11. `Gibt es in hier in der Nähe noch eine ältere Brücken?`
12. `Sagen Sie mir einen Weg zum Parkhaus!`
13. `Beschreiben Sie mir einen Weg zum Parkhaus!`
14. `Schlag mir eine Route zum Philosophenweg vor!`
15. `Berechne mir die schnellste Route zum SchloB!`
16. `Gibt es am Schloss einen Bahnhof?`
17. `Gibt es einen Bahnhof am Schloss?`
18. `Wie komme ich am schnellsten zum Bahnhof?`
19. `Wo geht es hier zum Zentralen Betriebshof der HSB?`
20. `Wo geht es hier zum zentralen Betriebshof der HSB?`

21. Wie kommt man von hier zur Sparkasse?

22. Wie kommt man zur Kirche?

23. Welchen Weg kann ich zum Uni-Platz nehmen?

24. Welchen Weg muß ich nehmen, wenn ich zurück zur Haltestelle will?

25. Welchen Weg muß ich nehmen, wenn ich zum nächsten Briefkasten will.

26. Welchen Weg muß ich nehmen, wenn ich zur nächsten Bäckerei will.

27. Welchen Weg muß ich nehmen, wenn ich zum Postamt will?

28. Ich suche einen Briefkasten.

29. Ich suche das Schloß.

30. Ich suche ein Parkhaus.

31. Ich suche das Zentrum.

32. Bring mich zu einem Arzt!

33. Den Weg zu einer Tankstelle, bitte!

34. Zur nächsten Toilette,bitte !

35. Wie komme ich zu diesem Turm?

36. Wie komme ich zu dieser großen Halle?

37. Ist es noch weit von hier bis zum Uniplatz?

38. Ist der Uniplatz sehr weit entfernt?

39. Wie lange braucht man zu Fuß in die Innenstadt?

40. Wo bin ich?

41. Wo bin ich hier?

42. Wo bin ich hier jetzt angekommen?

43. Hilfe!

44. Gib mir meinen Standort?

45. Wo befinde ich mich?

46. Ist das die alte Brücke?

47. Bin ich jetzt auf dem Karlsplatz?

48. Wie heißt dieser Ort?

49. Wie heißt dieser Platz?

50. Welchen Namen hat dieses Denkmal?

51. Wer war der Erbauer dieses Hauses?

52. Wer ist zur Zeit Bürgermeister?

53. Wer war der Architekt dieses häßlichen Gebäudes?

54. Wann wurde die Stadtmauer abgerissen?

55. Wie heißt dieses Gebäude?

56. Was gibt's zum Schloss?

57. Schlossinfo?

58. Info zum Schloss!

59. Informationen zur Alten Brücke, bitte!

60. Gibt es hierzu Informationen?

61. Gibt es zu diesem Turm Informationen?

62. Hast Du was zu diesem Turm?

63. Weißt Du etwas über die alte Brücke?

64. Was weißt Du über das Hotel Ritter?

65. Geben Sie mir Informationen zur Hauptstrasse!

66. Ist das die alte Brücke?

Anhang B

Nutzernafragen nach der Filterung (Stufen 1–3)

1. <QUESTION><PATH><DESTINATION='<LEX='SCHLOSS'>'>
2. <DESTINATION='<LEX='SCHLOSS'>'>
3. <QUESTION><GET_TO><-I-><DESTINATION='<LEX='SCHLOSS'>'>
4. <LEX='POST'>
5. <-I-><DESTINATION='<LEX='HBF'>'>
6. <QUESTION><LEX='HAUPTSTRAßE'>
7. <GET_TO><-I-><VICINITY><DESTINATION='Brücke'>
8. <QUESTION><GET_TO><-I-><VICINITY><DESTINATION='<LEX='OLD BRIDGE'>'>
9. <PATH><VICINITY><DESTINATION='<LEX='OLD BRIDGE'>'>
10. <LEX='OLD BRIDGE'>
11. <VICINITY><VICINITY><LEX='OLD BRIDGE'>
12. <PATH><DESTINATION='Parkhaus'>
13. <PATH><DESTINATION='Parkhaus'>
14. <PATH><DESTINATION='Philosophenweg vor'>
15. <SPEED='quick'><PATH><DESTINATION='<LEX='SCHLOSS'>'>
16. <LEX='SCHLOSS'><LEX='HBF'>
17. <LEX='HBF'><LEX='SCHLOSS'>
18. <QUESTION><GET_TO><-I-><SPEED='quick'><DESTINATION='<LEX='HBF'>'>
19. <VICINITY><DESTINATION='Zentralen Betriebshof der HSB'>
20. <VICINITY><DESTINATION='zentralen Betriebshof der HSB'>
21. <QUESTION><GET_TO><VICINITY><DESTINATION='Sparkasse'>

22. <QUESTION><GET_TO><DESTINATION='Kirche'>

23. <PATH><-I-><DESTINATION='<LEX='UNIPLATZ'> nehmen'>

24. <PATH><-I-><-I-><DESTINATION='Haltestelle will'>

25. <PATH><-I-><-I-><DESTINATION='<NEXT>en <LEX='BRIEFKASTEN'> will'>

26. <PATH><-I-><-I-><DESTINATION='<NEXT>en <LEX='BAECKEREI'> will'>

27. <PATH><-I-><-I-><DESTINATION='<LEX='POST'> will'>

28. <-I-><GET_TO><LEX='BRIEFKASTEN'>

29. <-I-><GET_TO><LEX='SCHLOSS'>

30. <-I-><GET_TO><NP=' Parkhaus'>

31. <-I-><GET_TO><NP=' Zentrum'>

32. <-I-><DESTINATION='Arzt'>

33. <PATH><DESTINATION='Tankstelle'>

34. <NEXT><NP=' Toilette'>

35. <QUESTION><GET_TO><-I-><DESTINATION='Turm'>

36. <QUESTION><GET_TO><-I-><DESTINATION='großen Halle'>

37. <QUESTION><VICINITY><DESTINATION='<LEX='UNIPLATZ'>'>

38. <QUESTION><LEX='UNIPLATZ'>

39. <QUESTION><NP=' Innenstadt'>

40. <LOCATION>

41. <LOCATION>

42. <LOCATION><VICINITY><GET_TO>

43. <LOCATION>

44. <LOCATION>

45. <LOCATION><-I->

46. <INFORMATION='vicinity'><LEX='OLD BRIDGE'>

47. <-I-><NP=' Karlsplatz'>

48. <QUESTION><INFORMATION='name'><NP=' Ort'>

49. <QUESTION><INFORMATION='name'><NP=' Platz'>

50. <INFORMATION='name'><NP=' Denkmal'>

51. <INFORMATION='who'><NP=' Erbauer'><NP=' Hauses'>

52. <INFORMATION='who'><DESTINATION='Zeit Bürgermeister'>

53. <INFORMATION='who'><INFORMATION='Architekt'><NP=' häßlichen Gebäudes'>

54. <INFORMATION='time'><NP=' Stadtmauer'>

55. <QUESTION><INFORMATION='name'><NP=' Gebäude'>

56. <QUESTION><DESTINATION='<LEX='SCHLOSS'>'>

57. <LEX='SCHLOSS'><INFORMATION='default'>

58. <INFORMATION='default'><DESTINATION='<LEX='SCHLOSS'>'>

59. <INFORMATION='default'><DESTINATION='<LEX='OLD BRIDGE'>'>

60. <VICINITY><INFORMATION='default'>

61. <DESTINATION='Turm <INFORMATION='default'> '>

62. <QUESTION><DESTINATION='Turm'>

63. <INFORMATION='default'><LEX='OLD BRIDGE'>

64. <QUESTION><INFORMATION='default'><NP=' Hotel Ritter'>

65. <INFORMATION='default'><DESTINATION='<LEX='HAUPTSTRAßE'>'>

66. <INFORMATION='vicinity'><LEX='OLD BRIDGE'>

Anhang C

Flex Quellcode

1. Filter

```
/* Thomas Cottmann     */

/* Parser strata 0->1 */


/* This stratum operates on the word level only! */

/* certain words are identified and a lexical look up takes place */


%option noyywrap


%x ICH WQ


PUNCT [\?!":.,\- <>()[\]{}']


BRIEFKASTEN Briefkasten

BAECKEREI Bäcker|Bäckerei

UNIPLATZ (Uni[ -]?[pP]latz)|Universitätspl(a|ae|ä)tz

SCHLOSS Schlos|Schloss|Schloß|SchloB

RATHAUS Rath(a|ae|ä)us

HAUPTSTRASSE Haupt(str|strasse|straße)

POST  "Haupt"?[pP]ost(amt)?

HBF Hbf|("Haupt"?[bB]ahnhof)

HBF-HD ("Heidelberger "(Hbf|("Haupt"?[bB]ahnhof)))|((Hbf|("Haupt"?[bB]ahnhof))" Heidelberg")

OLDBRIDGE (a|ae|ä|A)"lte"n?r?e?" Brücke"n?


/* ------------------------------------------------------------------------------ */


PATH ([^A-z][Ww]eg)|([Rr]oute[^A-z])
```

```
GETTO ([Kk]omme?t?)|suche
NEAR [nN]ähe|hier
NEXT nächst

QUICK schnell|"am schnellsten"|"möglichst schnell"
SHORT kürzeste

LOCATION_1 ([Ss]tandort)|([Hh]ilfe)
LOCATION_2 [Ww]o" ".*" ich"
LOCATION_3 [Ww]o" ".*" ich hier?"

INFORMATION_0 INFORMATION|INFO|INFOS|INFORMATIONEN|information|info|infos|informationen|Information|Info|Infos|Informationen
INFORMATION_1 NAME|([nN]ame)|heisst|heißt
INFORMATION_2 [Ww]ann" "
INFORMATION_3 [Ww]er" "
INFORMATION_4 Architekt|Gründer|gegründet|(erbaute?)|(errichtete?)
INFORMATION_5 [Ww]issen|[Ww]eisst|[Ww]eißt
INFORMATION_6 ^[Ii]st({PUNCT}|\n)(das|dies)({PUNCT}|\n)

%%


{BRIEFKASTEN} printf("<LEX='BRIEFKASTEN'>");
{BAECKEREI} printf("<LEX='BAECKEREI'>");
{UNIPLATZ} printf("<LEX='UNIPLATZ'>");
{SCHLOSS} printf("<LEX='SCHLOSS'>");
{RATHAUS} printf("<LEX='RATHAUS'>");
{HAUPTSTRASSE} printf("<LEX='HAUPTSTRAßE'>");
{POST} printf("<LEX='POST'>");
{HBF} printf("<LEX='HBF'>");
{HBF-HD} printf("<LEX='HEIDELBERG HBF'>");
{OLDBRIDGE} printf("<LEX='OLD BRIDGE'>");

{PATH} printf("<PATH>");
{GETTO} printf("<GET_TO>");
{NEAR} printf("<VICINITY>");
{NEXT} printf("<NEXT>");
```

```
{QUICK} printf("<SPEED='quick'>");

{SHORT} printf("<SPEED='short'>");



{LOCATION_1} printf("<LOCATION>");

{LOCATION_2} printf("<LOCATION>");

{LOCATION_3} printf("<LOCATION>");


{INFORMATION_0} printf("<INFORMATION='default'> ");

{INFORMATION_1} printf("<INFORMATION='name'>");

{INFORMATION_2} printf("<INFORMATION='time'>");

{INFORMATION_3} printf("<INFORMATION='who'>");

{INFORMATION_4} printf("<INFORMATION='%s'>",yytext);

{INFORMATION_5} printf("<INFORMATION='default'> ");

{INFORMATION_6} printf("<INFORMATION='vicinity'> ");


 /* Ich */

^[Ii]ch BEGIN(ICH);

({PUNCT}[Ii]ch)|({PUNCT}mich) BEGIN(ICH);


<ICH>{PUNCT}|\n {printf("<-I->%s",yytext);BEGIN(0);}

<ICH>[A-z] printf("ich %s",yytext);



 /* Questions */

^[Ii]st BEGIN(WQ);

^[Ww](as|ie) BEGIN(WQ);

{PUNCT}[Ww](as|ie) BEGIN(WQ);

<WQ>{PUNCT}|\n {printf("<QUESTION>%s",yytext);BEGIN(0);}


 /* Bitte */


[, ]*[Bb]itte" "?[.?!]  printf(".");

[, ]*[Bb]itte" "?[.?!]?\n printf(".\n");

[Bb]itte ;


%%
```

```
main (){
   printf("<MESSAGE='stratum 0 passed'>\n");
  yylex();
}
```

2. Filter

```
/* Thomas Cottmann               */
/* Parser strata 1->2           */

/* This stratum looks for destinations and NPs with articles */

%option noyywrap

%x zum det1 np1

PUNCT [\?!":.,\- <>()[\]{}']

ZUM ([Zz]u[mr]|[Nn]ach)" "

DER er|es|em|en|ie|er
DIE ie|er|en
DAS as|es|em|er|en|ie
EIN in|iner|ine|ines|inen|inem
DIES ieser|iesen|iesem|iese|ieses
JENER ener|enes|enem|enen|ene|ener

DIGIT [0-9]

NUMBER eins|zwei|drei|vier|fünf|sechs|sieben|acht|neun|zehn|elf|zwölf|zwanzig|dreißig|vierzig|fünfzig|sechzig|siebzig|achtzig|ne

PRAEP in|im|an|nach|vor|bei|zu|hinter|während|über|unter



%%

{ZUM} {printf("%s",yytext);BEGIN(zum);}
zu[ \t]+((eine[mr])|(diese[rm]))[ \t]+ {printf("%s",yytext);BEGIN(zum);}
```

```
/* zum BLABLA ? oder [new line] wird als Zielangabe verstanden */
<zum>[^"?"\n]+ {printf("<DESTINATION='%s'>",yytext); BEGIN(0);}



/* Identifikation von NPs über Schlüsselwörter */

/* Artikel und Demonstrativpronomina */

/* Nach Satzzeichen oder Space folgt kleines oder grosses 'D' */
{PUNCT}[Dd] {printf("%s",yytext);BEGIN(det1);}
/* Am Zeilenanfang steht kleines oder grosses 'D' */
^[Dd] {printf("%s",yytext);BEGIN(det1);}

{PUNCT}[Ee] {printf("%s",yytext);BEGIN(det1);}
^[Ee] {printf("%s",yytext);BEGIN(det1);}

{PUNCT}[Jj] {printf("%s",yytext);BEGIN(det1);}
^[Jj] {printf("%s",yytext);BEGIN(det1);}


/* ------------------------------------------*/

<det1>{DER} {printf("%s",yytext); BEGIN(np1);}
<det1>{DIE} {printf("%s",yytext); BEGIN(np1);}
<det1>{DAS} {printf("%s",yytext); BEGIN(np1);}
<det1>{EIN} {printf("%s",yytext); BEGIN(np1);}
<det1>{DIES} {printf("%s",yytext); BEGIN(np1);}
<det1>{JENER} {printf("%s",yytext); BEGIN(np1);}

/* falscher Alarm */
<det1>.|\n {printf("%s",yytext); BEGIN(0);}

/* Blanks eine Zahl, ein  Adjektiv (Wort in lower mit min. 3 Buchstaben)
   und mehrere Substantive (1.upper Rest lower) */
<np1>[ \t\n]+(({DIGIT}|{NUMBER})+[ \t\n]+)?([a-züöäß]{3,}[ \t\n]+)?([ \t\n]*[A-ZÜÖÄ][a-züöäß]+("-"[A-ZÜÖÄ][a-züöäß]+)?)+
{printf("<NP='%s'>",yytext);BEGIN(0);}
```

```
 /* für Relativsätze wie: die, die den ... */

<np1>{PUNCT}[Dd] {printf("%s",yytext);BEGIN(det1);}
<np1>^[Dd] {printf("%s",yytext);BEGIN(det1);}

<np1>{PUNCT}[Ee] {printf("%s",yytext);BEGIN(det1);}
<np1>^[Ee] {printf("%s",yytext);BEGIN(det1);}

<np1>{PUNCT}[Jj] {printf("%s",yytext);BEGIN(det1);}
<np1>^[Jj] {printf("%s",yytext);BEGIN(det1);}
 /* -----------------------------------------*/

 /* wenn kein Leerzeichen, Tab oder CR folgt, Fälle wie: deswegen */
<np1>[^ \t\n] {printf("%s",yytext); BEGIN(0);}

 /* nächsten bzw. <NEXT> */

"<NEXT>"[a-z]* {printf("%s",yytext);BEGIN(np1);}

%%

main (){
  printf("<MESSAGE='stratum 1 passed'>\n");
  yylex();
}
```

3. Filter

```
/* Thomas Cottmann               */
/* Parser strata 2->3           */


/* This stratum cleans up: only tags leave this stage */

 int iteration=0;

%option noyywrap

%x inside
```

```
PUNCT [\.!?;,:]

%%

[^<\n] ; /* delete everything (before a tag) */
"<" {printf("%s",yytext);iteration++;BEGIN(inside);} /* jump to inside tag area */
^\n ; /* delete lines with just a line break */
\n printf("%s",yytext);

<inside>{PUNCT} ;
<inside>[A-z ='0-9] printf(yytext);
<inside>">" {printf("%s",yytext);iteration--;if (iteration==0) BEGIN(0); else BEGIN(inside);}
<inside>"<" {printf("%s",yytext);iteration++;}

%%

main (){
  printf("<MESSAGE='stratum 2 passed'>\n");
  yylex();
}
```

Anhang D

C Quellcode des finalen Transducers

```
/* This is the last stratum of the parsing process, */

/* here the final decisions are made and results    */

/* are handed over to the next part of the system.   */



#include <stdio.h>

#include <stdlib.h>

#include <string.h>

/* 150 lines: 120 characters each line for user query*/

#define LINES 150

#define CHARS 121

#define OBJECT 100

#define TOP 150

#define HEAVY 60

#define LIGHT 40

#define STANDARD 10

/* input filename */

#define INPUT_FILE "3.txt"


  /* The following structs are mainly used for a rating based analysis. */

  /* Additionally, they store the values to be handed over to the next module */
```

```
  /* user query: information */
struct information_struct
{
  int rating;
  char query_obj[51];
  char query_type[51];
} information;

  /* user query: location */
struct location_struct
{
  int rating;
} location;

  /* user query: routing */
struct route_struct
{
  int rating;
  char dest_obj[51];
  char on_route_obj[51];
  char speed[11];
} route;

 /* tag existence analysis */

struct tag_struct
{
  int destination;
  int get_to;
  int i;
  int information;
  int lex;
  int location;
  int next;
  int np;
  int path;
  int speed;
  int question;
```

```
  int vicinity;

} tags;

char query[LINES][CHARS]; /* Text with queries */


/* This counts the tags in each sentence */
void tag_count(char *str);

/* find matches strings in input */
int match(char *input, char *search);

/* determines end of a tag in a given string */
int tag_end(char *str);

/* reads user inputs from file */
int readfile(char *fname,int lines,int chars);
/* filename, number of lines, chars per line */

/* finite state machines to process the scores */
int machines(int item);

/* determines parameter for route and information query */
void r_param(int item);
void i_param(int item);

/* ------------------------------------------------------------------ */
/*                            BEGIN OF MAIN                           */
/* ------------------------------------------------------------------ */
int main(void)
{

  int rcode=0; /* return-code */
  int t,tt,winner,temp;

char inputf[]=INPUT_FILE; /* input filename for testing on large data */
```

```
/* --------- */

for(t=0; t<LINES; ++t)
  for(tt=0; tt<CHARS;++tt) query[t][tt]=0;

rcode=readfile(inputf,LINES,CHARS);

if (rcode==0)

{

  for(t=3; t<LINES; ++t) /* calculate and display results */
                  /* skipping first 3 lines with messages */
{
  if (query[t][0]!=0) /* suppresses display of empty user queries */
 {
    /* reset overall ratings */
   route.rating=0;
   location.rating=0;
   information.rating=0;
   winner=0; /* 0=default 1=route 2=location 3=information */

   printf("\n(%d) ",t-2); /* print query # */
   tag_count(query[t]); /* get tag score */
   temp=machines(t); /* finite state machines to process the scores */
   if (!temp) /* not a singele machine has reached the final state */
     {
      route.rating=0;
      location.rating=0;
      information.rating=0;
     }
   printf("\nRatings: route(%d) location(%d) information(%d)",route.rating,location.rating,information.rating);

   /* Who has the maximum rating? */
```

```
    /* compare the first pair */
    if (route.rating>location.rating) winner=1;
    else winner=2;
    /* compare biggest of first pair with third */
    if (winner==1)
       if (route.rating>information.rating) winner=1;
       else winner=3;
    else
       if (location.rating>information.rating) winner=2;
       else winner=3;

    if((route.rating==0)&&(location.rating==0)&&(information.rating==0)) winner=0;

    printf("\nFinal result: ");
    /* Display results and determine parameters */
    switch(winner){

    case 1: r_param(t); printf("route (%s, %s, %s)",route.dest_obj,route.on_route_obj,route.speed);break;
    case 2: printf("location (current)");break;
    case 3: i_param(t);printf("information (%s, %s)",information.query_obj,information.query_type);break;
    default: printf("default");
    }

    printf("\n----------------------\n");





 }
}



} /* End of if: rcode */
else printf("\n-> Ein Fehler beim Einlesen der Daten ist aufgetreten! <-\n");
```

```
 return(0);
}
/* ------------------------------------------------------------------ */
/*                          END OF MAIN                               */
/* ------------------------------------------------------------------ */

/* This counts the tags in each sentence */
void tag_count(char *str)
{
 char input[201];
 char *p;

 strcpy(input,str); /* a copy, for later recursions on input */

  /* reset the score */
 tags.destination=0;
 tags.get_to=0;
 tags.i=0;
 tags.information=0;
 tags.lex=0;
 tags.location=0;
 tags.next=0;
 tags.np=0;
 tags.path=0;
 tags.speed=0;
 tags.question=0;
 tags.vicinity=0;

 tags.destination=match(input,"<DESTINATION='");
 tags.get_to=match(input,"<GET_TO>");
 tags.i=match(input,"<-I->");
 tags.information=match(input,"<INFORMATION='");
 tags.lex=match(input,"<LEX='");
 tags.location=match(input,"<LOCATION>");
 tags.next=match(input,"<NEXT>");
 tags.np=match(input,"<NP='");
```

```
 tags.path=match(input,"<PATH>");
 tags.speed=match(input,"<SPEED='");
 tags.question=match(input,"<QUESTION>");
 tags.vicinity=match(input,"<VICINITY>");


 /* output of the tag struct */
 printf("filtered user query:\n\n%s\n",input);

 if (tags.destination>0) printf("%d <DESTINATION>\n",tags.destination);
 if (tags.get_to>0)  printf("%d <GET_TO>\n",tags.get_to);
 if (tags.i>0)  printf("%d <-I->\n",tags.i);
 if (tags.information>0)  printf("%d <INFORMATION>\n",tags.information);
 if (tags.lex>0)  printf("%d <LEX>\n",tags.lex);
 if (tags.location>0)  printf("%d <LOCATION>\n",tags.location);
 if (tags.next>0)  printf("%d <NEXT>\n",tags.next);
 if (tags.np>0)  printf("%d <NP>\n",tags.np);
 if (tags.path>0)  printf("%d <PATH>\n",tags.path);
 if (tags.speed>0)  printf("%d <SPEED>\n",tags.speed);
 if (tags.question>0)  printf("%d <QUESTION>\n",tags.question);
 if (tags.vicinity>0)  printf("%d <VICINITY>\n",tags.vicinity);

}

/* END OF tag_count */

match(char *input, char *search)
{
 int count=0;
 int t;
 char *str;
 char *p;

 str=input; /* a copy, just in case :-) */

 do /* check for multiple matches */
 {
   p = strstr(str,search);
```

```
   if (p!=NULL) /* match found */
  {
    count++; /* increase match count */
    p++; /* cut off first element of matched string */
   str=p;
  }
 }while (p!=NULL); /* no further matches */

 return(count);
}



/* reads user inputs from file */
int readfile(char *fname, int lines,int chars)
{

FILE *fp;
char ch;
int current_line=0, position=0;



if((fp=fopen(fname, "r"))==NULL) return(1);

ch=getc(fp); /* read one character */

while ( (ch!=EOF) && (current_line<lines) )
{
  if (position==(chars-1)) position=0;
  query[current_line][position]=ch;
  position++;
  if (ch=='\n')
  {
    current_line++;
    position=0;
  }
  ch = getc(fp);
}
```

```
fclose(fp);

return(0);
}



/* finite state machines to process the scores */
int machines(int item)
{
  int safety=0; /* against infinite loops */
  int automaton=1;




 goto start; /* goto start state */



 /* start state */
 start:
{

safety++; if (safety>150) return(0);

/* switch between different automatons */
switch(automaton){

  case 1:
         automaton++;
         if (tags.location)
          {location.rating=location.rating+TOP;goto end;}
         break;

  case 2:
         automaton++;
         if (tags.information)
          {information.rating=information.rating+TOP;goto end;}
         break;
```

```
case 3:
       automaton++;
       if (tags.path)
        {route.rating=route.rating+TOP;goto end;}
       break;

case 4:
       automaton++;
       if (tags.destination)
        {goto node_1;}
       break;


case 5:
       automaton++;
       if (tags.get_to)
        {goto node_2;}
       break;

case 6:
       automaton++;
       if (tags.question)
        {goto node_3;}
       break;

case 7:
       automaton++;
       if (tags.vicinity)
        {goto node_4;}
       break;

case 8:
       automaton++;
       if (tags.next)
        {route.rating=route.rating+HEAVY;goto end;}
```

```
        break;


  case 9:
        automaton++;
        if (tags.i)
         {location.rating=location.rating+STANDARD;goto end;}
        break;



  case 10:
        automaton=1;
        if ((tags.lex)||(tags.np))
         {route.rating=route.rating+STANDARD;goto end;}
        break;





   }
 goto start;
}


 node_1:
{
 if (tags.get_to)
  {route.rating=route.rating+(2*HEAVY);goto end;}
 else goto node_2;
}


 node_2:
{
 if (tags.i)
  {route.rating=route.rating+(2*HEAVY);goto end;}
 else
  {route.rating=route.rating+LIGHT;goto end;}
}


 node_3:
```

```
{

 if ((tags.np)||(tags.lex))
  {route.rating=route.rating+LIGHT;goto end;}
 else
  {information.rating=information.rating+STANDARD;goto end;}
}

 node_4:
{
 if ((tags.np)||(tags.lex))
  {route.rating=route.rating+LIGHT;goto end;}
 else
  {route.rating=route.rating+STANDARD;goto end;}
}


 /* final state */
 end:return(1);

}


/* determines end of a tag in a given string */
int tag_end(char *str)
{

int length,count=0;
int recursion=1;


length=strlen(str);

do
 {
   if (*str=='<') recursion++;
   if (*str=='>') recursion--;
```

```
   /* printf("\n%c %d ",*str,recursion);*/

   if (recursion) {str++;count++;}
 }
while( (recursion) && (count<length) );

count--; /* for deletion of '> of destination tag */
/* printf("\n#%d ",count);*/
return(count);

}



/* determines parameter for route query */
void r_param(int item)
{

int length,i,t;
char str[OBJECT],temp[OBJECT];
char *p;

str[OBJECT-1]=0;
temp[OBJECT-1]=0;

 /* default parameters */
 strcpy(route.dest_obj,"error");
 strcpy(route.on_route_obj,"default");
 strcpy(route.speed,"default");

 /* is there a <DESTINATION> tag? */
if (tags.destination)
  {
  p = strstr(query[item],"<DESTINATION='");
  p=p+14; /* jump over <DESTINATION=' */
  length=tag_end(p);
  memmove(str,p,length);
  str[length]=0; /* end string properly */
  if (!match(str,"<")) /* there are no other tags in the destination field */
```

```
  {
   printf(" (parameter guessed!)  ");
   /* jump over heading spaces */
   for (i=0; (isspace(str[i]))&&(i<OBJECT-1);i++);
   /* a trivial approach, just take the first word and dump the rest */
   for(t=0;(t<OBJECT-1-i);t++) /* loop through input */
     {
       if (isspace(str[i+t])) break; /* space found exit loop */
       temp[t]=str[i+t]; /* no space, copy char into temp */
     }
   temp[t]=0; /* end string properly */
   strcpy(route.dest_obj,temp);
  }
else /* there are other tags in the destination field */
  {

   if (match(str,"<LEX=")) /* start of <LEX> */
     {
      p = strstr(str,"<LEX=");
      p=p+6; /* jump over <LEX=' */
      for(t=0;t<OBJECT-1;t++)
       {
        if (*p=='>')break;
        str[t]=*p;
        p++;
       }
      str[t-1]=0; /* overwrite ' */
      strcpy(route.dest_obj,str);
     } /* end of <LEX> */


   if (match(str,"<NP=")) /* start of <NP> */
     {
      p = strstr(str,"<NP=");
      p=p+6; /* jump over <LEX=' */
      /* jump over heading spaces */
      while(isspace(*p))
       {
```

```
          if (*p==0) break;
          p++;
         }
        for(t=0;t<OBJECT-1;t++)
         {
          if (*p=='>')break;
          str[t]=*p;
          p++;
          if (isspace(*p)) {p++; break;} /* complex NPs chopped */
         }
        str[t-1]=0; /* overwrite ' */
        strcpy(route.dest_obj,str);
       } /* end of <NP> */



    }


   }/* End of <DESTINATION> tag analysis */
else
  {
/* look for stand alone tags/potential objects */


  if (match(query[item],"<LEX=")) /* start of <LEX> */
       {
        p = strstr(query[item],"<LEX=");
        p=p+6; /* jump over <LEX=' */
        for(t=0;t<OBJECT-1;t++)
         {
          if (*p=='>')break;
          str[t]=*p;
          p++;
         }
        str[t-1]=0; /* overwrite ' */
        strcpy(route.dest_obj,str);
       } /* end of <LEX> */



     if (match(query[item],"<NP=")) /* start of <NP> */
```

```
       {
        p = strstr(query[item],"<NP=");
        p=p+6; /* jump over <LEX=' */
        /* jump over heading spaces */
        while(isspace(*p))
         {
          if (*p==0) break;
          p++;
         }
        for(t=0;t<OBJECT-1;t++)
         {
          if (*p=='>')break;
          str[t]=*p;
          p++;
          if (isspace(*p)) {p++; break;} /* complex NPs chopped */
         }
        str[t-1]=0; /* overwrite ' */
        strcpy(route.dest_obj,str);
       } /* end of <NP> */

/* end of look for stand alone tags */
  }

/* test for general tags */

if (tags.speed)
        {
         p = strstr(query[item],"<SPEED='");
         p=p+8; /* jump over <SPEED=' */
         length=tag_end(p);
         memmove(str,p,length);
         str[length]=0; /* end string properly */
         strcpy(route.speed,str);
        }

} /* r_param */

/* ------------------------------------------------------------------ */
```

```
/* ------------------------------------------------------------------- */
/* ------------------------------------------------------------------- */

/* determines parameter for information query */
/* up tp the moment basically the same code and strategy as in r_param */
void i_param(int item)
{

int length,i,t;
char str[OBJECT],temp[OBJECT];
char *p;

str[OBJECT-1]=0;
temp[OBJECT-1]=0;



 /* default parameters */
 strcpy(information.query_obj,"default");
 strcpy(information.query_type,"default");

 /* is there a <DESTINATION> tag? */
if (tags.destination)
  {
  p = strstr(query[item],"<DESTINATION='");
  p=p+14; /* jump over <DESTINATION=' */
  length=tag_end(p);
  memmove(str,p,length);
  str[length]=0; /* end string properly */
  if (!match(str,"<")) /* there are no other tags in the destination field */
    {
     printf(" (parameter guessed!)  ");
     /* jump over heading spaces */
     for (i=0; (isspace(str[i]))&&(i<OBJECT-1);i++);
     /* a trivial approach, just take the first word and dump the rest */
     for(t=0;(t<OBJECT-1-i);t++) /* loop through input */
       {
         if (isspace(str[i+t])) break; /* space found exit loop */
         temp[t]=str[i+t]; /* no space, copy char into temp */
```

```
    }
   temp[t]=0; /* end string properly */
   strcpy(information.query_obj,temp);
  }
else /* there are other tags in the destination field */
  {

   if (match(str,"<LEX=")) /* start of <LEX> */
     {
      p = strstr(str,"<LEX=");
      p=p+6; /* jump over <LEX=' */
      for(t=0;t<OBJECT-1;t++)
       {
        if (*p=='>')break;
        str[t]=*p;
        p++;
       }
      str[t-1]=0; /* overwrite ' */
      strcpy(information.query_obj,str);
     } /* end of <LEX> */


   if (match(str,"<NP=")) /* start of <NP> */
     {
      p = strstr(str,"<NP=");
      p=p+6; /* jump over <LEX=' */
      /* jump over heading spaces */
      while(isspace(*p))
       {
        if (*p==0) break;
        p++;
       }
      for(t=0;t<OBJECT-1;t++)
       {
        if (*p=='>') break;
        str[t]=*p;
        p++;
        if (isspace(*p)) {p++; break;} /* complex NPs chopped */
```

```
         }
        str[t-1]=0; /* overwrite ' */
        strcpy(information.query_obj,str);
       } /* end of <NP> */



    }


   }/* End of <DESTINATION> tag analysis */
else
  {
/* look for stand alone tags/potential objects */


  if (match(query[item],"<LEX=")) /* start of <LEX> */
       {
        p = strstr(query[item],"<LEX=");
        p=p+6; /* jump over <LEX=' */
        for(t=0;t<OBJECT-1;t++)
         {
          if (*p=='>')break;
          str[t]=*p;
          p++;
         }
        str[t-1]=0; /* overwrite ' */
        strcpy(information.query_obj,str);
       } /* end of <LEX> */



     if (match(query[item],"<NP=")) /* start of <NP> */
       {
        p = strstr(query[item],"<NP=");
        p=p+6; /* jump over <LEX=' */
        /* jump over heading spaces */
        while(isspace(*p))
         {
          if (*p==0) break;
          p++;
         }
```

```
        for(t=0;t<OBJECT-1;t++)
         {
          if (*p=='>') break;
          str[t]=*p;
          p++;
          if (isspace(*p)) {p++; break;} /* complex NPs chopped */
         }
        str[t-1]=0; /* overwrite ' */
        strcpy(information.query_obj,str);
       } /* end of <NP> */

/* end of look for stand alone tags */
  }

/* test for general tags */

if (tags.information)
        {
         p = strstr(query[item],"<INFORMATION='");
         p=p+14; /* jump over <INFORMATION=' */
         length=tag_end(p);
         memmove(str,p,length);
         str[length]=0; /* end string properly */
         strcpy(information.query_type,str);
        }



} /* i_param */
```

Anhang E

Inhalt der CD (`readme.txt`)

```
************************************************************
************************************************************
** Inhalt der CD zu                                       **
** Parsing Modul für ein Touristen-Informationssystem     **
** Lehrstuhl für Computerlinguistik, Universität Heidelberg **
** SS 2001                                                **
** Thomas Cottmann - Blücherstraße 3 - D 69115 Heidelberg **
************************************************************
************************************************************

readme.txt - diese Datei :-)

anfrage.txt - das Originalkorpus
count2.lex - ein Lex-Beispiel zum Zählen von Ausdrücken
decision - exe des 4. Programmteils
decision.0 - backup des 4. Programmteils: 0
decision.1 - backup des 4. Programmteils: 1
decision.2 - backup des 4. Programmteils: 2
decision.final - backup des 4. Programmteils: final version
decision.c - C Quellcode des 4. Programmteils
distribution.eps - distribution of tags (Enapsulated PostScript)
distribution.sds - dstribution of tags (StarOffice5)
docu.lyx - Dokumentation (LYX)
```

docu.txt - Dokumentation (TXT)

docu.ps - Dokumentation (PostScript)

filter_chain.eps - Programmteile (Enapsulated PostScript)

information.eps - tag score information (Enapsulated PostScript)

information.sds - tag score information (StarOffice5)

information_p.eps - relative tag score information (Enapsulated PostScript)

information_p.sds - relative tag score information (StarOffice5)

keyboard.lex - ein Lex-Beispiel zum Zählen lesen von stdin

location.eps - tag score location (Enapsulated PostScript)

location.sds - tag score location (StarOffice)

loaction_p.eps - relative tag score location (Enapsulated PostScript)

location_p.sds - relative tag score location (StarOffice)

machines.sda - Transducers (StarOffice5)

new - Shellscript zum Löschen nach LEX Benutzung

np0.txt - Nominalphrasen

np1.txt - Nominalphrasen nach Filterung

parse0 - exe des 1. Programmteils

parse0.lex - LEX-Code des 1. Programmteils

parse0a.lex - LEX-Code des 1. Programmteils mit stdin Eingabefunktion

parse1 - exe des 2. Programmteils

parse1.lex - LEX-Code des 2. Programmteils

parse2 - exe des 3. Programmteils

parse2.lex - LEX-Code des 3. Programmteils

parser - Shellscript für alle Teile

parser1 - Shellscript für alle Teile mit Tastatureingabe

points1.eps - Minitransducer (Enapsulated PostScript)

result.txt - das geparste Korpus

route.eps - tag score route (Enapsulated PostScript)

route.sds - tag score route (StarOffice5)

route_p.eps - relative tag score route (Enapsulated PostScript)

route_p.sds - relative tag score route (StarOffice5)

scores_p.eps - relative tag score route location information (Enapsulated PostScript)

scores_p.sds - relative tag score route location information(StarOffice5) stage1.txt - Ausgabe von parse0

stage2.txt - Ausgabe von parse1

stage3.txt - Ausgabe von parse2

stage3a.txt - Ausgabe von parse2 (modifiziert)

stage3b.txt - Ausgabe von parse2 (bereinigt)

stage4.ps - Ausgabe des Parsers in Spalten (PostScript)

stage4.sdw - Ausgabe des Parsers in Spalten (StarOffice5)

stage4.txt - Ausgabe des Parsers (TXT)

start - Shellsript zum Arbeiten mit LEX, braucht input.txt

transducer.eps - der finale Transducer

vergleich.ps - Vergleich von Anfrage vor und nach der Filterung (PostScript)

vergleich.sdw - Vergleich von Anfrage vor und nach der Filterung (StarOffice5)