LudwigV

Natürlich Sprachliche Wissensbasis

Bernd Thomas
Charlottenstr.45
56077 Koblenz

Michael Kühn
Cusanusstr.26
56073 Koblenz

6. Juni 1995

Zusammenfassung

Die vorliegende Ausarbeitung zu dem Programm LudwigV, war eine Programmieraufgabe im Rahmen des sprachorientierten KI Forschung Seminars bei Herrn Prof. Dr. Hess. Das Programm ist in Sicstus Prolog 2.1 geschrieben.

Kapitel 1

Einleitung

Die Aufgabe die uns gestellt wurde, lautete: *'Ein kleines System zum Aufbauen und Abfragen eines Informationsystems zu erstellen'*.

Ausgehend von dieser Aufgabenstellung, verfolgten wir das Ziel eine möglichst natürlich sprachliche Schnittstelle zwischen System und Benutzer zu realisieren. Desweiteren lag es in unserem Interesse das Aufbauen der Wissensbasis und die Anfragen an diese in Form eines natürlich sprachlichen Dialogs zu implementieren. Damit dieser Dialog möglichst formlos gehalten werden kann, haben wir das System zusätzlich mit der Eigenschaft ausgestattet anaphorische Textkohärenzen zu berücksichtigen.

Kapitel 2

Syntax

2.1 Die Module

Unser Ziel war es das Programm so in Module zu splitten, daß man es auch in Zukunft um neue Fähigkeiten leicht erweitern kann. So stellt die Erweiterung der DCG-Grammatik um *Features* und *Slash-Mechanismus* in Verbindung mit dem implementierten Compiler fuer die *DCG-SF Grammatik* ein Werkzeug dar, mit dem man etwas anspruchsvollere natürlich sprachliche Eingaben analysieren (parsen) kann. An der morphologischen Analyse sind folgende Module beteiligt:

- Das Lexikon
- Das Flexions Lexikon
- Lexikonschnittstelle inlex(Wort,GrammatischeKategorien,Lemma)

Die Bearbeitung der Grammatik und das Parsen der Benutzereingabe wird von folgenden zwei Modulen übernommen:

- Der DCG-SF Compiler

- Die DCG-SF Grammatik

2.2 Das Lexikon

Das Lexikon ist wie folgt aufgebaut:

- **Feature Definition:**
  `l_verb :: [subc, vform, per, num, temp, perfx].`
- **Stammformeintrag:**
  `lexikon(schreib,schreib,l_verb(subc:trans..temp:praes),vflex_reg).`
  `lexikon(schrieb,schreib,l_verb(subc:trans..temp:praet),vflex_praet_pl_e).`
  `lexikon(geschrieb,schreib,l_verb(subc:trans..perfx:hab),part2_en).`
- **Festformeintrag:**
  `vflexikon(hat,hab,l_verb(subc:hv..vform:fin..per:3..num:sg..temp:pras)).`

Die *Feature Definition* legt die verfügbaren grammatischen Kategorien bezüglich einer Wortart fest, wobei nicht nur die *Features*, die im Lexikon verwendet werden definiert werden, sondern auch solche die erst im *Flexions Lexikon* näher bestimmt werden (z.B.: bei Verben: Numerus, Person etc.). Der Wortart-Bezeichner (hier: l_verb) ist frei wählbar, muss jedoch, einmal eingeführt, beibehalten werden.

Die *Features* müssen in ihrer Schriftform nicht mit den, in der Grammatik verwendeten *Features*, übereinstimmen. Die nötige Referenz wird in der Grammatik durch die *Lexikon-Interface-Regel* hergestellt.

Die Werte, der *Features*, sind frei wählbar, sollten jedoch der grammatischen Kategorie entsprechend sinnvoll gewählt werden, und müssen konsistent beibehalten werden.

Ein *Stammformeintrag* besteht aus folgenden Argumenten:

- **dem Stamm:**
  `lexikon(`*STAMM*`,Lemma,wortart(Features),Flex.Kat).`
  Der `STAMM` muss den Features oder der Flexionskategorie entsprechen. So müssen aufgrund der Stammumlautung bei Verben (Präsens/Präteritum/2 o. 3 Person Singular) mehrere Lexikoneinträge aufgenommen werden. Ähnliches gilt für andere Wortarten.

- **das Lemma:**
  `lexikon(Stamm,`*LEMMA*`,wortart(Features),Flex.Kat.).`
  Das `LEMMA` stellt eine Reduktion der Flexionsformen des Wortes auf eine Grundform dar.
  Sollte also für jeden Eintrag bzgl. eines Wortes gleich sein, da auf das Lemma in der Grammatik zugegriefen wird und bei der Bildung der semantischen Terme wichtig ist.

- **die Features:**
  `lexikon(Stamm,Lemma,`*wortart(FEATURES)*`,Flex.Kat.).`
  Die `FEATURES` sind abhängig von der Wortart und werden in der *Feature Definition* bzgl. jeder Wortart festgelegt. Das Belegen grammatischer Kategorien erfolgt durch *Feature:Wert z.B.: kasus:nom.*

- **Flexionskategorie:**
  `lexikon(Stamm,Lemma,wortart(Features),`*FLEX.KAT.*`).`
  Alle Wörter, die als Stammformeinträge abgelegt werden, benötigen eine morphologische Information über ihre möglichen Wortformen.
  `FLEX.KAT.` gibt die entsprechende Kategorie an nach der das Wort flektiert wird. Die verschiedenen Flexionskategorien sind im *Flexions Lexikon* abgelegt.

Die *Festformeinträge* unterscheiden sich von den Stammformeinträgen nur geringfügig dadurch, daß die Angabe der Flexionskategorie entfällt, und in der Regel mehr Featurebestimmungen besitzen. Dies erklärt sich dadurch, daß die morphologische Analyse mittels der Flexionskategorie und die damit verbundenen Featurebestimmungen, bei Festformeinträgen entfällt und die Features explizit abgelegt werden.

2.3 Das Flexions Lexikon

Das *Flexions Lexikon*[1] bildet die Grundlage der morphologischen Analyse, hier sind die möglichen Flexionen der im *Lexikon* eingetragenen Wörter nach Flexionskategorien abgelegt. Der Aufbau eines Flexions Lexikoneintrags:

```
flex([EndungAsciCode‡Stamm],Endung,Stamm,wortart(Features),Flex.Kat.).
```

mit EndungAsciCode=[110,101] entspricht Endung=en

Die jeweiligen Argumente des Eintrags sind in ihrer Bedeutung gleich denen der Lexikoneinträge. Wobei *EndungAsciCode* die umgedrehte Flexionsendung des Wortes in AsciCodes darstellt.

2.4 Die Lexikonschnittstelle inlex

Die Schnittstelle zum Lexikon bildet das Prologprädikat

```
inlex(Wort,wortart(Features),Lemma).
```

Inlex[2] übernimmt die Lexikonsuche und morphologische Analyse eines Wortes. Wobei seine Arbeitsweise wie folgt beschrieben werden kann:

- Suche im *Flexions Lexikon* einen Eintrag, entsprechend der Wortart, der genau die Wortendung besitzt wie das zu bestimmende Wort.
- Suche bzgl. der Informationen aus dem *Flexions Lexikon* einen passenden Eintrag im *Lexikon*
- wenn ein entsprechender Lexikoneintrag existiert, dann übergebe die *Features* und das *Lemma* an `inlex`.

Dieser ganze Vorgang wird fast ausschließlich vom Unifikationsalgorithmus des Prologinterpreters übernommen. Damit die Endungen der Wörter bearbeitet werden können arbeiten wir im *Flexions Lexikon* mit dem umgedreh-

[1] Datei flexion.pl
[2] in Datei `ludwigv.pl`

ten Wort in Listennotation, was das bestimmen der Wortendung und des Wortstamms erleichtert.

2.5 Die DCG-SF Grammatik

Struktur einer DCG-SF Regel
Eine DCG-SF Regel besitzt folgende Syntax:

```
regelname(Featureauzählung, Argument1, Argument2, ..) -->
                                DCG-SF-Regel.
```

Die Featureaufzählung und die Argumente sind fakultativ, wobei die ArgumenteX in ihrer Anzahl beliebig gewählt werden können. Auf der rechten Seite der Regel können beliebig viele DCG-SF-Regeln oder Prologprädikate auftretten, wobei die Prologprädikate in {} stehen müssen. Bezeichner die mit Grossbuchstaben beginnen sind Variable.

Konstituenten Feature Definition
Unsere Intention war es die DCG-Grammatiken so zu erweitern, daß wir eine Phrasenstrukturgrammatik benutzten konnten, deren Konstituenten spezielle, zuvor definierte Merkmale (Features) besitzen. Dies wird durch die, den Grammatikregeln vorangestellten *Feature Definitionen* erreicht.

```
vp :: [person, numerus, verbform, tempus]
```

Diese Beispiel-Definiton würde den Gebrauch einer `vp`-Konstituenten mit den Merkmalen `person, numerus, verbform` und `tempus` innerhalb der Grammatik zulassen. Eine mögliche Regel könnte dann folgende Form annehmen:

```
vp(person:P,numerus:N,verbform:fin) -->
  v(person:P,numerus:N,verbform:fin),
   np(kasus:akk).
```

Hier würde die Konstituente `v` die Person und den Numerus der `vp`-

Konstituente erben, und die `vp` Regel würde nur ausgeführt wenn die Verbform der `vp` *fin* wäre. An die `np`-Konstituente würde die Bedingung gestellt, daß sie im Akkusativ steht.

Lexikon-Interface

Da wie zuvor beschrieben[3] die Features der Lexikoneinträge nicht die gleiche Schriftform besitzen müssen, wie die in der Grammatik verwendeten Features, muss eine Schnittstelle zwischen den lexikalischen Konstituenten der Grammatik und den Lexikoneinträgen bereitgestellt werden. Dies übernimmt die Lexikoninterfaceregel, die für jede lexikalische Konstituente erstellt werden muss. Ein Beispiel:

```
v(numerus:N..person:P..tempus:T) --> [Verb],
{ inlex(Verb,l_verb(num:N..per:P..temp:T),Lemma) }
```

Es wird eine lexikalische Konstituente `v` eingeführt die in ein Wort (genauer: Zeichenkette `[Verb]`) übergeht, mit den Merkmalen Numerus, Person und Tempus. Wichtig ist hier, daß ein Agreement zwischen LexikonFeatures und GrammatikFeatures erreicht wird, durch die Variablen `N, P,` und `T`. Erfüllt wird die Regel nur dann, wenn daß momentan bearbeitete Wort, den `inlex` Aufruf mit den gesetzten oder geforderten Merkmalen erfüllt. Auch bei den Lexikoninterfaceregeln können Prologprädikate auf der rechten Seite der Regel auftreten, jedoch nur in der Form :

```
lex_konst(GramFeatures, Arg1, .. ,ArgX) ) --> [WORT],
    { inlex(WORT, wortart(LexFeatures), Lemma), Prologprädikat }.
```

Der Slash-Mechanismus

Da es Konstituenten innerhalb der synstaktischen Struktur eines Satzes gibt, die ihre Position ändern, wäre es nützlich und angemessen einen Mechanismus innerhalb des Regelwerkes zu besitzen, der dieses Phänomen behandelt. So kann bei dem Satz:

[Der Junge]$_{\texttt{NPnom}}$ *schlägt [den Mann]*$_{\texttt{NPakk}}$.

die `NPakk` in den Topic des Satzes gestellt werden, also:

[Den Mann]$_{\texttt{NPakk}}$ *schlägt [der Junge]*$_{\texttt{NPnom}}$.

Der Slash-Mechanismus ist ein Hilfsmittel solche *Bewegungen* zu berücksich-

[3] siehe Kapitel 2.1 Lexikon

tigen. Beispiel:

```
vp(Features) --> v(Features),
                 np(kas:nom),
                 np(kas:akk).

s(typ:dek) --> np(NPFeatures),
               vp(vform:fin) / np(NPFeatures).
```

Die `s`-Regel würde durch den Gebrauch des Slash-Operators `/` solche Bewegungen, der Nominalphrasen zulassen. Die Arbeitsweise des Slash-Operators, ist wie folgt zu beschreiben: Der `/` wird auf die `vp`-Regel angewendet, was bedeutet, daß aus der `vp`-Konstituente eine `np`-Konstituente eliminiert werden soll, die genau die gleichen Features (`NPFeatures`) besitzt wie die `np`-Konstituente die im Topic, also Spitzenstellung des Satzes steht. Der Slash *klaut* sich also eine Konstituente-`np(NPFeatures)` aus der `vp`-Konstituente, er eliminiert aber nicht irgendeine Konstituente, sondern nur so eine, die in Konstituententyp und Features mit den geforderten Eigenschaften übereinstimmt. Findet er in der *geshlashten* `vp` keine entsprechende `np`, so ist die Regel nicht erfüllbar.
Um eine Epsilon-Ableitung, eine leere Ableitung, zu erhalten, ist es zusätzlich nötig eine spezielle Slash-Regel aufzunehmen:

```
np(Features) / np(Features) --> [].
```

Sie bedeutet, daß eine Konstituente in die leere Ableitung übergeht, wenn sie mit genau den SlashFeatures versehen ist, die sie selber als GrammatikFeatures besitzt. Das ist genau was wir erreichen wollten. Solch eine Slash-Regel muss für jeden Konstituententyp erstellt werden, der geslashed werden soll. Auch hier ist es wieder möglich auf der rechten Seite der Regel Prologprädikate zu benutzen, jedoch auch nur in eingeschränkter Form:

`np(Features) / np(Features) --> {` *Prologprädikat* `}, [].`

Die Semantischen Terme
Da wir die syntaktisch analysierten Sätze, in eine semantische Repräsentation überführen mußten, für die spätere Auswertung der Satzsemantik und zur Behandlung der Wissensbasis, haben wir neben den Features ein Argument mit in die Regeln aufgenommen, das für die jeweilige Konstituente die semantische Repräsentation enthält. So haben die von uns gebrauchten DCG-SF Regeln die allgemeine Form:

```
konst(Features,SemantischerTerm) --> konstx(Featuresx,SemTermx).
```

Ein Beispiel:

```
  vp(typ:T..sem:VX, (SemN, v(VX, Lem, r(VX, role1, NX)))) -->
            v(sub:intra..lem:Lem..typ:T, Lem),
             { agree(T,P,N) }, % Agreement Subjekt und Verb
             n(kas:nom..per:P..num:N..sem:NX, SemN).
```

Die Komponierung der semantischen Terme, innerhalb der Grammatik, richtet sich nach den Forderungen die das *Semantik-Evaluations* Modul, an die Form der Terme stellt. Die Zusammensetzung und Bedeutung der semantischen Terme wird in Kapitel 3 behandelt.

2.6 Der DCG-SF Compiler

Der DCG-SF Compiler[4] übersetzt die DCG-SF Regeln in Prologprädikate, die dann zur syntaktischen Analyse und Bildung der semantischen Terme aufgerufen werden.

Struktur der DCG-SF Regel als Prologregel
Die Struktur einer DCG-Regel in Prolog:

```
s --> np, vp. ⇒
s(Text,TRest) :- np(Text,Text1), vp(Text1,TRest).
```

Der DCG-SF Compiler erzeugt aus den DCG-SF Regeln Prologprädikate

[4] Datei dcg_comp2.pl

der Form:

```
s(Feature,Ax) --> vp(FeatureVP,Ay), np(FeatureNP,Az).⇒
s(SlashListe,SlashListeRest,Feature,Ax,s(SynStVP,SynStNP),Text,TRest):-
  vp(SlashListe,SlashListe1,FeatureVP,Ay,SynStVP,Text,Text1)
  np(SlashListe1,SlashListeRest,FeatureNP,Az,SynStNP,Text1,TRest).
```

Mit folgenden Argumenten:

SlashListe: sie enthält die zu *slashende* Konstituente.

SlashListeX(Rest): wenn der *Slash* mit der Konstituenten übereinstimmt, dann ist `SlashListeX(Rest)` die *SlashListe* ohne den *Slash*. Die aufgerufene Konstituente ist also *geslashed* worden.

FeatureXX: die grammatikalischen Merkmale der Konstituente[5].

Az: beliebige Argumente, beliebige Anzahl.

SynSTXX: die syntaktische Struktur der Konstituente.
Sie stellt eine Baumstruktur in Termnotation dar. Bsp.: s(FeaS,np(FeaNP,det(FeaDET,*der*,FeaLex),n(FeaN,*mann*,FeaLex)), vp(FeaVP,v(FeaV,*schläft*,FeaLex)))

Text1: zu analysierender Text.

TRest: nicht analysierter Text.

Diese Argumente werden teilweise übernommen(*Az*) und teilweise vom Compiler eingefügt(*SlashListe, SynSTXX ..*). Die so erhaltene Regelstruktur entspricht dann der einer DCG-Regel, die dann in ein Prologprädikat[6] überführt wird.

Featurebehandlung des Compilers
Zur Bearbeitung der speziellen Featureschreibweise ist es nötig einen Übersetzer zu besitzen, der die Featureaufzählungen in eine interne Repräsentation umwandelt. Die interne Repräsentation muss bei unserer Implementation für jede Wortart die gleiche sein, daß heißt egal ob eine Featureaufzählung

[5] siehe Kapitel 2.5
[6] geschieht i.a. durch built in predicate: expand_term

(kas:nom..gen:fem) nur zwei Elemente besitzt, in ihrer internen Repräsentation wird ihr eine Liste mit genau so vielen Elementen zugeordnet wie in der jeweiligen *Feature Definition* aufgezählt sind (Bsp.: `(kas:nom..gen:fem)` → `[_,_,nom,_,fem]`). Dies ist eine Aufgabe[7] die von dem *DCG-SF Compiler* übernommen wird. Der tiefere Sinn ist der, daß für jede Wortart gleiche Feature-Listenlängen existieren und die Features immer an den richtigenPositionen in den Listen stehen, unabhängig von der Reihenfolge der Aufzählung ,so kann der Unifikationsalgorithmus auf diese Listen angewendet werden.

Behandlung des Slash

Innerhalb der DCG-SF Regeln wird der Slash als Operator benutzt, in den compilierten Regeln jedoch, haben wir für ihn zwei Argumente im Regelkopf jedes übersetzten Prologprädikats besetzt. Diese Argumente enthalten eine Liste, deren Inhalt ein oder mehrere Slashs sind. Wobei die erste der beiden Listen[8] die noch zu *bearbeitenden* Slashs vor Aufruf einer Konstituenten beinhaltet, und die zweite Liste[9] die noch zu bearbeitenden Slashs nach dem Aufruf der Konstituenten.

Die SlashListe ist wie ein Stapel organisiert, es kann also nur dann eine leere Ableitung einer Konstituenten erfolgen, wenn der entsprechende Slash, bzgl. dieser Konstituente zu letzt in die SlashListe eingefügt wurde.
Beispiel:

```
s --> np(kas:nom), vp2(typ:fin) / np(kas:nom).
vp2(typ:fin) --> np(kas:akk), vp(typ:fin) / np(kas:akk).
vp(typ:fin) --> v(typ:fin), np(kas:akk), np(kas:nom).
```

In diesem Beispiel würde also zuerst eine leere Ableitung der Nominalphrase im Akkusativ erfolgen, und dann erst würde die `np(kas:nom)` geslashed. Würde die Reihenfolge der Slashs vertauscht, also in der `s` Regel eine `np(kas:akk)` geslashed, so wäre es nicht möglich mit den angegebenen Regeln einen Satz der Form: *Der Mann schlaeft* abzuleiten.

[7] die Featurebeahndlung wird von den Prologprädikaten `feat_defs(Type,Feat)`, zur Behandlung der Feature Definitionen und `feat_term(Typ, Feats, FeatListe)`, zur Behandlung der Featureaufzählung in der Datei `dcg_comp2.pl` übernommen

[8] siehe SlashlisteX, Abschnitt Struktur der DCG-SF Regel

[9] siehe SlashListeX(Rest), Abschnitt Struktur der DCG-SF Regel

Anmerkung
Da das Lexikon und das Flexions-Lexikon auch die Featureschreibweise(`Feature:Wert..Feature:Wert`) benutzen, ist es erforderlich auch sie zu compilieren. Der Compiler verändert dabei nichts an der Struktur der Fakten oder Prädikate, er ersetzt lediglich die Featureschreibweise durch die beschriebene Listennotation. Dies gilt auch für verwendete Fakten und Prädikate innerhalb der Grammatik, sie werden übernommen, wobei nur die Featureschreibweise übersetzt wird.

Kapitel 3

Semantische Evaluierung

Der Output der syntaktischen Analyse ist eine semantische Repräsentation des Eingabesatzes, die aus einem Indikator für den Satztyp und einer Reihe von semantischen Termen besteht.
Diese werden nun an ein ein Modul `evaluate` weitergereicht, das die semantischen Terme auswertet, indem es versucht, die Angaben (bei Deklarativsätzen) in die Wissensbasis abzuspeichern oder — bei Fragesätzen — zu bestätigen.

Unser Anspruch war hierbei nicht, ein bestimmtes semantisches Modell zu implementieren.
Vielmehr haben wir die Struktur und Behandlung der semantischen Repräsentation der Aufgabenstellung und unserer eigenen Zielsetzung angepaßt [1].
Zu unseren Zielen gehörten eine möglichst flexible Wissensrepräsentation und die Möglichkeit, in der natürlich-sprachlichen Eingabe auf bereits Eingegebenes Bezug nehmen zu können.

Um dieses in unserem bescheidenen Rahmen verwirklichen zu können, unterscheiden wir folgende Typen von semantischen Termen:

[1] ... und dafür bei bestehenden Modellen Anleihen gemacht.

- quantifizierte Terme (bestimmte, unbestimmte und gefragte),
- Namen,
- Anaphern,
- Ereignisse und
- Relationen.

Jede dieser semantischen Strukturen beinhaltet in der einen oder anderen Form einen „Referenzindex'", über den die Strukturen miteinander verknüpft werden und Wissen in der Datenbank abgelegt wird.

3.1 Referenzindex und anaphorisches Genus

Der Referenzindex steht für einen ganze Zahl, die jedem im Text eingeführten Objekt eindeutig zugeordnet wird, gleich ob es sich um eine Person, eine Sache, einen Zeitpunkt oder ein Ereignis handelt.

Unter dieser Zahl wird jede Information zu diesem Referenten in der Wissensbasis abgelegt.
Wird später auf den Referenten in einem Term Bezug genommen, so erhält der semantische Term den abgelegten Index in seiner Referenzvariable.

So verweisen *Beethoven* und *der Komponist* in

Beispiel 1 *Beethoven*$_1$ *ist ein Komponist*

und

Beispiel 2 *Der Komponist*$_1$ *wurde 1770 geboren*

auf denselben Referenten (hier Index 1).

Zusätzlich zu dem Referenzindex wird das syntaktische Genus in den Termen angegeben, mit dem auf den Referenten zugegriffen wurde.
Dies ist nötig, damit Anaphern nur auf den Referenten im letzten Satz verweisen können, der in seinem Genus mit dem der Anapher übereinstimmt.

Schwierigkeiten ergeben sich hier bei Eigennamen, die an sich kein syntaktisches Genus besitzen.
Wir behelfen uns, indem wir davon ausgehen, daß Eigennamen bei ihrem ersten Auftreten das Genus von einem anderen Term erben, welches dann in der Datenbank abgelegt wird.
So weisen wir in Beispiel 1 dem Referenten 1 das syntaktische Genus von *ein Komponist* zu (also maskulin).

Schließlich haben wir das „klassische'"Genus (mas, fem, neut) um die Werte *loc* und *tem* (*loc*al und *tem*poral) erweitert, um den Gebrauch von anaphorischen Ausdrücken wie *da* oder *dort* zu ermöglichen.

3.2 Quantifizierte Terme

Ein quantifizierter Term entsteht bei der syntaktischen Analyse von Nominal phrasen.
Seine allgemeine Struktur ist `q(`*Quantor*`,` *Index*`,` *Genus*`,` *Term*`)`. *Index* und *Genus* sind die oben beschriebenen Angaben Referenzindex und anaphorisches Genus, *Term* ist wieder ein semantischer Term, der für den Referenten *Index* gelten soll.

Quantor kann in unserer Implementierung einen der drei Werte `d`, `e` und `w` annehmen. Damit werden bestimmte, unbestimmte und gefragte Nominalphrasen erfaßt.

3.2.1 Bestimmt quantifizierter Term

Der bestimmt quantifizierte Term wird in der syntaktischen Analyse durch den definiten Artikel *der** erzeugt (z.B. *der Komponist* `q(d, X, p(mas), r(X,komponist,true))`).

Grundsätzlich gehen wir davon aus, daß durch einen definit quantifizierten Term auf ein vorher Erwähntes Bezug genommen werden soll. Dafür muß der Referent durch *Term* eindeutig in der Wissensbasis referiert werden.

Werden für den Referenten mehrere mögliche Kandidaten gefunden, wird der d-quantifizierte Ausdruck nicht aufgelöst, d.h. die semantische Evaluierung insgesamt schlägt fehl.

Gibt es gar keinen passenden Kandidaten, wird ein neuer Referenzindex erstellt und dem d-Term zugewiesen, wenn dieser Term in einem Deklarativsatz auftritt. Dadurch können insbesondere definite Zeitangaben, die vorher gar nicht aufgetreten sind, behandelt werden.

3.2.2 Unbestimmt quantifizierter Term

Unbestimmt (oder e-) quantifizierte Terme stammen von Nominalphrasen mit dem unbestimmten Artikel *ein**. Diese bezeichnen in Fragesätzen irgendeinen Referenten, auf den die angegebenen Prädikate zutreffen.

In Deklarativsätzen wird durch den e-quantifizierten Ausdruck grundsätzlich ein neuer Referent eingeführt, mit der Ausnahme, daß der Referenzindex schon mit einem Wert instantiiert ist.
Dieser Fall tritt bei Copula-Konstruktionen wie Beispiel 1 ein. Hier wird schon in der syntaktischen Analyse der Referenzindex von *ein Komponist* mit dem von *Beethoven* gleichgesetzt, da dies ja gerade der Intention des Copula-Verbs *sein* entspricht.

3.2.3 Gefragter Term

Der gefragte Term ist unsere semantische Repräsentation für Nominalphrasen, die ein W-Fragewort enthalten, als Determiner (*welcher**) oder als vollständige NP (*wer* oder auch *wann*).
Wir fassen wegen der syntaktischen Parallelen den gefragten Ausdruck daher auch als w-quantifizierten Term auf, obwohl der Begriff Quantor hier vielleicht unglücklich gewählt wird.

Auch die semantische Behandlung ist eher einfach gehalten. Da der Index des gefragten Terms in W-Fragesätzen im Satztyp mit angegeben ist, kann der w-Term in Fragen wie ein e-quantifizierter bearbeitet werden.

In Deklarativsätzen werden w-Terme einfach übergangen (da sie hier ja eigentlich gar nicht auftreten sollten).

Hier wäre eine genauere Unterscheidung sicher angebracht. Für unsere Zwecke hielten wir diese Bearbeitung für ausreichend.

3.3 Namen

Die semantische Repräsentation von Eigennamen ist bei unserer Behandlung mit einigen Schwierigkeiten verbunden, da sie von sich aus kein eigenes syntaktisches Genus besitzen.

Wir nehmen das Genus dennoch in den semantischen Term mit auf, um es evt. doch mit abspeichern zu können. Daher hat z.B. der Name *Beethoven* folgende Struktur:

`n(`*Index*`, `*Genus*`, beethoven).`

In der Wissensbasis wird dieser Ausdruck in der Form `r(`*Index*`, name, beethoven)` und `r(`*Index*`, genus, `*Genus*`)` abgelegt.

3.4 Anaphern

Den semantischen Typ Anapher haben wir eingeführt, um in unserem Rahmen so etwas wie Textkohärenz verwirklichen zu können.
Unter diesen Begriff fallen bei uns Personalpronomen, aber auch adverbiale Proformen wie *dort*. Die Struktur eines anaphorischen Termes ist `p(`*Index*`,` *Genus*`)`.

Ihre Aufgabe ist, auf im Vorsatz erwähnte Referenten zuzugreifen. Zu diesem Zweck werden bei semantischen Evaluierung alle in einem Satz auftretende Referenten mit ihrem Genus zwischengespeichert.
Wird nun im nachfolgenden Satz eine Anapher verwendet, so sucht sie sich einen Referenten aus dem letzten Satz, der in seinem Genus mit ihrem übereinstimmt. Natürlich wird dieser Referent dann wieder in den Kurzzeitspeicher aufgenommen.

Dadurch ist es möglich, über mehrere Sätze hinweg anaphorisch auf einen Referenten zuzugreifen.

3.5 Ereignisse

Der semantische Typ Ereignis `v(`*Index*`,` *Lemma*`,` *Term*`)` repräsentiert in unserer Analyse die Verben, die eine eigene Behandlung erfahren.

Sie besitzen wie die anderen Typen einen Referenzindex, der sich hier nicht auf eine Person o.ä. bezieht, sondern auf ein bestimmtes Ereignis, das durch *Lemma* näher klassifiziert wird, z.B. *schlaf*, *sterb* etc.

In *Term* steht eine Reihe von Angaben, die sich auf die an dem Ereignis beteiligten Referenten bezieht. In Beispiel 2 auf Seite 14 sieht der gesamte Ausdruck für das Ereignis dann so aus (angenommen, daß *Beethoven* den Index 1 und der Zeitpunkt *1770* den Index 3 besitzt) :

```
v(X, gebor, (r(X, role2, 1), r(X, roleT, 3))).
```

Dabei steht `role2` für den zweiten Partizipanten und `roleT` für den Zeitpunkt, an dem das Ereignis stattfand.
Diese Angaben werden u.a. gebraucht, um für ein Ereignis entscheiden zu können, ob es sich um ein bereits bekanntes oder ein neues handelt.

3.6 Bekanntes und Neues

Die Unterscheidung zwischen Bekanntem und Neuem spielt bei Deklarativsätzen eine entscheidende Rolle, da nur neue Informationen in der Wissensbasis abgespeichert werden sollen.

Um dies zu erreichen, werden in dem `evaluate`-Aufruf für Deklarativ-Sätze erst einmal alle möglichen Evaluierungen eines Satzes gebildet, von denen dann diejenige genommen wird, die die wenigsten *neuen* Informationen enthält.

Dieses Verfahren spiegelt unserer Meinung nach die menschlichen Strategie wider, neue Informationen so weit wie möglich mit bereits bekannten zu verknüpfen.

Nehmen wir zu unserem Dialog folgenden Satz hinzu:

Beispiel 3 *Beethoven wurde in Bonn geboren.*

Dann wird bei der semantischen Evaluierung festgestellt, daß dieses Ereignis bereits genannt wurde und die vorhandenen Angaben sich nicht widersprechen. Also wird als einzig neue Information festgehalten, daß der Ort des Ereignisses in Bonn war (dies wird von dem Prädikat `known/8` erledigt).

Natürlich könnte man auf ein Ereignis, das im letzten Satz beschrieben wurde, auch anaphorisch verweisen (was wir noch nicht implementiert ha-

ben): *Das geschah in Bonn*; dies ist jedoch nicht möglich, wenn das Ereignis nicht im letzten Satz auftauchte, sondern seine Erwähnung schon länger zurückliegt.

In dem Fall ist es wichtig, daß das Ereignis als schon bekannt erkannt wird und nur noch die neuen Angaben dazugelernt werden.

3.7 Fragen und Antworten

Im Gegensatz dazu gehen wir bei Fragesätzen davon aus, daß keine neuen Informationen dazukommen sollen, sondern daß die Angaben im Satz auf der Wissensbasis bestätigt werden müssen (bei Ja-Nein-Fragen) oder eine bestimmte Angabe gefunden werden soll.

Im Programm wird beides mit dem Prädikat `prove` realisiert.

Für Ergänzungsfragen besteht das Ergebnis aus einer Liste von Referenzindizes, die die Anfrage erfüllen. Diese müssen nun in einem zweiten Schritt nachbearbeitet werden, indem für sie semantische Terme gebaut werden, aus denen dann natürlich-sprachliche Antworten generiert werden können.

Dies wurde nur sehr rudimentär implementiert, insbesondere da es für die Antwort-Generierung sehr nützlich gewesen wäre, auf die ursprüngliche Syntax-Struktur zugreifen zu können.
Dies ist in unserer Implementierung nicht realisiert.

Kapitel 4

Fazit

Wir hatten nicht den Ehrgeiz (und hoffentlich auch nicht die Aufgabe), ein Dialogsystem zu entwickeln, daß mit allen möglichen Besonderheiten eines natürlich-sprachlichen Dialoges zurechtkommt.
Vielmehr ging es uns darum zu zeigen, welche sprachlichen Phänomene in unserem Rahmen realisiert werden konnten und wie eine geeignete Wissensrepräsentation für solch eine NL-Datenbank aussehen kann.

So werden in unserer Implementierung unterschiedliche sprachliche Erscheinungen wie unbekannte Eigennamen, Passiv-Transformationen, optionale Ort- und Zeitangaben auf der syntaktischen Ebene , und anaphorische Textkohärenz, bestimmte und unbestimmte Referenz, aber auch Ereignisse auf der semantischen Ebene behandelt.
Wir wollen nicht behaupten, daß diese Phänomene erschöpfend ausgearbeitet wurden; aber wir sind der Meinung zu jedem dieser Problemkreise einen für unsere Arbeit angemessenen Ansatz aufgezeigt zu haben.

Eine prinzipielle Einschränkung, die wir uns selbst auferlegten, war, daß die semantischen Repräsentationen nicht inhaltlich definiert werden sollten, so daß in unserer Implementierung keine Inferenzen der Art *„Wenn X ein Komponist ist, dann ist X ein Mensch“* möglich sind.
Dafür wäre es notwendig, die einzelnen semantischen Terme nicht in der

Grammatik festzulegen, sondern sie für jedes Wort aus dem Lexikon zu holen.
Am Algorithmus der semantischen Evaluierung müsste dafür nichts geändert werden.

Weiter auszuarbeiten ist auf jeden Fall die Abarbeitung der Fragesätze, die in unserer Implementierung etwas zu kurz kommt.
So ist es weder möglich, nach zwei oder mehr Angaben zu fragen (*„Wer hat was wann komponiert?“*), noch nach Ereignissen: *„Was geschah 1770?“*

Auch die Antwort-Generierung fiel etwas notdürftiger aus, als wir uns gewünscht hatten.
Ein Problem hier war, daß wir in unserer Implementierung nicht mehr auf die ursprüngliche Syntax-Struktur zurückgreifen konnten.
Das andere Problem bestand darin, einen allgemeinen Algorithmus aufzustellen, der die Dereferierung eines Referenzindexes so eindeutig wie möglich bewirkt.

Wir hatten also mit zwei verschiedenen Arten von Problemen zu kämpfen: einerseits mit solchen, die mit mehr Zeit ohne weiteres noch zu implementieren gewesen wären, andererseits mit Problemen, bei denen wir an prinzipielle Grenzen unserer Implementierung gestoßen sind.

Dennoch meinen wir, ein System geschrieben zu haben, das schon einige interessante Bereiche abdeckt und vor allem flexibel genug ist, um weiter ausgebaut zu werden.

Kapitel 5

Beispiel Dialog

Hier ein kleiner Dialog mit *LudwigV*, wobei die in Anführungstriche gesetzten Wörter, Namen entsprechen die, im Lexikon nicht enthalten sind.
Gross - Kleinschreibung und Interpunktionen werden ignoriert.

```
SICStus 2.1 #8: Tue Aug 31 13:48:26 MET DST 1993
| ?- [ludwigv].

---> DCG-COMPILER geladen <---
---> lade LEXIKON <---
Erfolgreich compiliert !
---> lade FLEXIONs LEXIKON <---
Erfolgreich compiliert !
---> lade Grammatik <---
Erfolgreich compiliert !
---> Semantik Evaluation geladen <---
---> BAUM-ROUTINEN geladen <---
---> Dialog Schleife geladen <---

yes
| ?- dialog.
```

```
 LudwigV  sokif seminar   B.Thomas M.Kuehn  16Feb94

[Ende mit "Ende."]

ja ? "Mozart" ist ein Komponist.

aha..wieder was dazugelernt:
r(1,komponist,true)
r(1,gen,p(mask))
r(1,name,mozart)

ja ? Er wurde 1756 in "Salzburg" geboren.

aha..wieder was dazugelernt:
r(4,roleL,3)
r(4,roleT,2)
r(4,role2,1)
r(4,event,gebor)
r(3,gen,loc)
r(3,name,salzburg)
r(2,punct,1756)

ja ? Er starb 1791 in "Wien".

aha..wieder was dazugelernt:
r(7,roleL,6)
r(7,roleT,5)
r(7,role1,1)
r(7,event,sterb)
r(6,gen,loc)
r(6,name,wien)
r(5,punct,1791)

ja ? Beethoven ist ein Komponist.

aha..wieder was dazugelernt:
r(8,komponist,true)
r(8,gen,p(mask))
```

```
r(8,name,beethoven)

ja ? Er schrieb eine Oper.

aha..wieder was dazugelernt:
r(10,role2,9)
r(10,role1,8)
r(10,event,schreib)
r(9,oper,true)

ja ? Er wurde 1770 in Bonn geboren.

aha..wieder was dazugelernt:
r(13,roleL,12)
r(13,roleT,11)
r(13,role2,8)
r(13,event,gebor)
r(12,gen,loc)
r(12,name,bonn)
r(11,punct,1770)

ja ? Er starb 1827 in Wien.

aha..wieder was dazugelernt:
r(15,roleL,6)
r(15,roleT,14)
r(15,role1,8)
r(15,event,sterb)
r(14,punct,1827)

ja ? 1789 wurde die Oper "Figaro" von Mozart in Wien geschrieben.

aha..wieder was dazugelernt:
r(18,roleL,6)
r(18,roleT,17)
r(18,role2,16)
r(18,role1,1)
r(18,event,schreib)
r(17,punct,1789)
```

```
r(16,gen,p(fem))
r(16,name,figaro)
r(16,oper,true)

ja ? Wer schrieb eine Oper?

[beethoven]
[mozart]

ja ? Wurde die Oper Figaro von Mozart geschrieben?

jau!

ja ? Beethoven schrieb die Oper "Fidelio".

aha..wieder was dazugelernt:
r(20,role2,19)
r(20,role1,8)
r(20,event,schreib)
r(19,gen,p(fem))
r(19,name,fidelio)
r(19,oper,true)

ja ? Was schrieb Mozart.

[figaro]

ja ? Mozart lebte in Wien.

aha..wieder was dazugelernt:
r(21,roleL,6)
r(21,role1,1)
r(21,event,leb)

ja ? Wo lebte Beethoven?

aeh..weiss ich nicht.

ja ? Beethoven lebte 1792 in Wien.
```

```
aha..wieder was dazugelernt:
r(23,roleL,6)
r(23,roleT,22)
r(23,role1,8)
r(23,event,leb)
r(22,punct,1792)

ja ? Wann lebte Beethoven?

[1792]

ja ? Wo lebt Beethoven 1792?

[wien]

ja ? Wer ist ein Komponist.

[mozart]
[beethoven]

ja ? Welcher Komponist wurde in Bonn geboren?

[beethoven]

ja ? Lebte er in Wien?

jau!

ja ? ende.

vielen dank. sie haben ein einfaches programm sehr gluecklich gemacht.
```