gefördert vom
Bundesministerium
für Forschung
und Technologie

Verbmobil
Verbundvorhaben

Parsing und Generierung in TrUG

Rudolf Caspari
Ludwig Schmid

Siemens AG, München

Report 40
Dezember 1994

Dezember 1994

Rudolf Caspari
Ludwig Schmid

Siemens AG
Otto-Hahn-Ring 6
81739 München

Tel.: (089) 636 - 49249
Fax: (089) 636 - 49802
e-mail: `Rudolf.Caspari@zfe.siemens.de`

Gehört zum Antragsabschnitt: 9 Generierung

Die vorliegende Arbeit wurde im Rahmen des Verbundvorhabens Verbmobil vom Bundesministerium für Forschung und Technologie (BMFT) unter dem Förderkennzeichen 01 IV 101 G gefördert. Die Verantwortung für den Inhalt dieser Arbeit liegt bei den Autoren.

Inhaltsverzeichnis

1 Einleitung

Dieser Bericht ist als Benutzerhandbuch für Anwender des TrUG-Systems gedacht, enthält aber auch über die Beschreibung der Funktionalität des Systems hinausgehende Information, welche insbesondere zur Motivation der in der Generierungskomponente eingeführten Begriffsbildungen dienen sollen.

Das TrUG-Ablaufsystem[1] besteht aus drei Bestandteilen:

- einer Komponente für die Analyse
- einer Komponente für die Synthese
- einer formalen Repräsentation als gemeinsame Schnittstelle zwischen Analyse und Synthese.

Durch die Analyse wird eine sprachliche Äußerung in eine formale Repräsentation der gemeinsamen Schnittstelle überführt. Bei der Synthese wird ausgehend von einer formalen Repräsentation eine sprachliche Äußerung erzeugt. Betrachtet man die formale Repräsentation als Sprache, so kann man sagen, daß das TrUG-Ablaufsystem die reversible Übersetzung von natürlicher Sprache in eine formale Sprache leistet.

Die Analyse(bzw.Synthese)-Komponente besteht aus einem Ablaufsystem für die Anwendung und Compilern für die Compilation von Grammatik, Lexikon und Transferlexikon. Die Komponenten werden mittels der folgenden UNIX-Shell-Scripts geladen.

2 Initialisierung der Ablaufsysteme

tas
lädt die Analysekomponente und die Synthesekomponente.
Dieses Kommando ist die bloße Konjunktion der Kommandos ta und ts.

ta
lädt die Analysekomponente.
Es werden zwei Fenster eröffnet:
ein Fenster für den Compiler und ein Fenster für das Ablaufsystem.
Die zur Initialisierung des Ablaufsystems erforderliche Software wird geladen.

ts
lädt die Synthesekomponente.
Es werden zwei Fenster eröffnet:
ein Fenster für den Compiler und ein Fenster für das Ablaufsystem.
Die zur Initialisierung des Ablaufsystems erforderliche Software wird geladen.

[1] Trace and Unification Grammar

3 Compilation von Grammatik und Lexikon

Grammatik und Lexikon werden im TrUG-Format erstellt und sind für Analyse und Synthese verfügbar, werden jedoch unterschiedlich compiliert.

3.1 Kommandos zur Compilation von Grammatik und Lexikon für die Analyse

tagc Grammatikdatei
übersetzt die Grammatikdatei ⟨Grammatikdatei⟩ in das Analyseformat und erzeugt die LR-Steuertabelle. Speichert das Ergebnis in zwei QOF-Dateien ⟨Grammatikdatei⟩.taf, ⟨Grammatikdatei⟩.lrt
und einer Textdatei ⟨Grammatikdatei⟩.yy

talc Lexikondatei
übersetzt die Lexikondatei ⟨Lexikondatei⟩ in das lose Analyseformat.
⟨Lexikondatei⟩ kann ein Lexikonteil oder ein vollständiges Lexikon sein.
Speichert das Ergebnis in der QOF-Datei ⟨Lexikondatei⟩.taf

3.2 Kommandos zum Compilieren von Grammatik und Lexikon für die Synthese

tsgc Grammatikdatei
übersetzt die Grammatikdatei ⟨Grammatikdatei⟩ in das Syntheseformat.
Speichert das Ergebnis in der QOF-Datei ⟨Grammatikdatei⟩.tsf
und der Textdatei ⟨Grammatikdatei⟩.gen

tslc Lexikondatei
übersetzt die Lexikondatei ⟨Lexikondatei⟩ in das Syntheseformat.
Speichert das Ergebnis in der QOF-Datei ⟨Lexikondatei⟩.tsf
und der Textdatei ⟨Lexikondatei⟩.sug

3.3 Optionen für die Compiler

Für die Compiler existieren die Optionen

-i IniDatei
Lädt vor dem Übersetzen die Datei ⟨IniDatei⟩

-l Ladename

speichert das Ergebnis in der Datei ⟨Ladename⟩.⟨Extension⟩

-t Taxonomie
lädt vor dem Übersetzen aus der Datei ⟨Taxonomie⟩ eine bereits übersetzte Taxonomie

Für die Lexikoncompiler existiert zusätzlich die Option
-k
Konvertiert das Lexikon beim Übersetzen in Kleinschrift

Die allgemeine Form der Compilerkommandos lautet:
talc [-k] [-i IniDatei] [-l Ladename] [-t Taxonomie] Lexikondatei
tslc [-k] [-i IniDatei] [-l Ladename] [-t Taxonomie] Lexikondatei
tsgc [-i IniDatei] [-l Ladename] [-t Taxonomie] Grammatikdatei
tagc [-O| -O2][-i IniDatei] [-l Ladename] [-t Taxonomie] Grammatikdatei
  Bei Option O wird die LALR(1)-Tabelle erzeugt statt der LR(0)-Tabelle.
  Es wird dynamischer Code erzeugt.
  Bei Option O2 wird die LALR(1)-Tabelle erzeugt statt der LR(0)-Tabelle.
  Es wird statischer Code erzeugt.

4 Formale Struktur von Grammatik und Lexikon

4.1 Struktur einer Grammatik

Eine Grammatik im TrUG-Format besteht aus einem Deklarationsteil und einem Regelteil. Der Deklarationsteil zerfällt seinerseits in einen allgemeinen und einen speziellen Teil wie nachstehend beschrieben.

Grammatik → Deklarationsteil   Grammatikregel*

Deklarationsteil → allgemeiner Deklarationsteil   spezieller Deklarationsteil

allgemeiner Deklarationsteil → Merkmalsdefinition*   Makrodefinition*   pragma-Term*

Merkmalsdefinition → KatSym => f(Attribut$_1$, ... Attribut$_N$)
Dabei gilt:
Attribut → einfaches Attribut | komplexes Attribut
einfaches Attribut → Attributname
komplexes Attribut → Attributname:Attributwert
Attributname → Atom
Attributwert → Atom
KatSym → Atom

Für einen Attributwert wird verlangt, daß er eine Definition besitzt.
Attributwert => f(Attribut$_1$, ... Attribut$_N$)   oder

Attributwert => $\{\mathrm{Atom}_1, \ldots \mathrm{Atom}_N\}$

Makrodefinition → Makrokopf short_for Bedingung*

Makrokopf → Functor | Functor($\mathrm{Variable}_1, \ldots, \mathrm{Variable}_N$)
Functor → Atom

pragma-Term →
pragma include(Dateiname) |
pragma warn_single_vars(+) |
pragma warn_single_vars(-) |
pragma tomita_special(Functor($\mathrm{ArgArt}_1, \ldots \mathrm{ArgArt}_N$) |
pragma tomita_return(StartSym, Spuren, SpezFs, AnaErg) |
pragma tomita_return(StartSym, Spuren, SpezFs, AnaErg) :- Rumpf |
pragma lazy_gen(Pfad)

spezieller Deklarationsteil → bounding_node-Definition* is_head_of-Definition* trace-Definition*

bounding_node-Definition → bounding_node(KatSym)

is_head_of-Definition → KatSym_1 is_head_of KatSym_2

trace-Definition →
trace(_, KatSym) | Bedingung*

Grammatikregel → linke Seite - - -> rechte Seite | Bedingung*
linke Seite → KatSym | KatSym:Index
rechte Seite → Kategorie*

Grammatikregel → linke Seite - - -> Kategorie + rechte Seite | Bedingung*
Das ist eine spezielle Grammatikregel zur Beschreibung der Landeposition für die Kopfbewegung.

Kategorie →
KatSym |
KatSym:Index |
KatSym_1<trace(ana, KatSym_2) |
KatSym_1<trace(var, KatSym_2)
Index → Atom

Bedingung → Bedingung ; Bedingung (Disjunktion von Bedingungen)
Bedingung → Gleichung | Inklusion | Makrokopf | SpezFn

Gleichung → $\mathrm{Kategorie}_1$:$\mathrm{Attributpfad}_1$ = $\mathrm{Kategorie}_2$:$\mathrm{Attributpfad}_2$
Gleichung → Kategorie:Attributpfad = Term

Attributpfad → Attributname
Attributpfad → Attributpfad:Attributpfad

Term → 'Functor(Arg_1,..., Arg_N)
Arg → Attributpfad | Atom

Inklusion → Kategorie:Attributpfad in {$Atom_1$,... $Atom_N$}

SpezFn ist eine mittels des pragmas tomita_special vorher definierte spezielle Form.

4.2 Beispiele und Erläuterungen

Beispiel von Merkmalsdefinitionen:
Für die Kategorie n werden das Merkmal morph und das Merkmal tree definiert.
n =⟩ f(morph:morph,tree).
morph =⟩ f(gen:gen , kas:kas , agr:agr).
gen =⟩ {mask , fem , neutr}.
kas =⟩ f(notgen:bin , notnom:bin , dir:bin).
agr =⟩ f(num:num , pers:pers).
num =⟩ {sg , pl}.
pers =⟩ {'1' , '2' , '3'}.
bin =⟩ {yes , no}.
n besitzt das einfache Attribut tree, sowie das komplexe Attribut morph. In der Definition von morph treten drei Attribute auf, von denen gen einen atomaren Wertebereich besitzt, wogegen kas und agr wiederum komplex sind.

Beispiel einer Makrodefinition:
nominativ(N) short_for N:notgen=yes , N:notnom=no , N:dir=yes.
Makros sind Abkürzungen für regelmäßig wiederkehrende Mengen von Bedingungen. Für eine modulare Grammatikentwicklung stellen sie ein komfortables und wichtiges Hilfsmittel dar.

Pragmas sind besondere Compileranweisungen.

pragma include(Dateiname)
bewirkt, daß an der Stelle, wo das Pragma steht, der Inhalt der Datei eingefügt wird. Include-Dateien dürfen geschachtelt sein, d.h. eine Include-Datei darf wieder Include-Pragmas enthalten.

pragma warn_single_vars(+) bzw. pragma warn_single_vars(-)
schaltet die Warnung vor einzelnen Variablen in den Leseeinheiten ein (+) bzw. aus (-). Die Warnung ist im Defaultfall eingeschaltet.

pragma tomita_special(Functor($ArgArt_1$, ... $ArgArt_N$))
definiert den angegebenen Ausdruck als spezielle Form. Spezielle Formen werden bei der Analyse in einer Liste gesammelt. Für jedes Argument ist dabei die Argumentart anzugeben. Mögliche Argumentarten sind:

cond	Argument ist TrUG-Bedingung (ohne spezielle Formen)
prog	Argument ist Prolog-Code mit TrUG-Ausdrücken
term	Argument ist ein TrUG-Term

Für die Initialisierung der flachen Semantik benötigt man folgende spezielle Formen:
pragma tomita_special(sem(term)).
pragma tomita_special(semdel(term)).
pragma tomita_special(semdelopt(term)).

pragma tomita_return(StartSym, Spuren, SpezFs, AnaErg) bzw.
pragma tomita_return(StartSym, Spuren, SpezFs, AnaErg) :- Rumpf
deklariert die Schnittstelle zwischen Prolog und TrUG. Für die Analyse wird festgelegt, wann sie erfolgreich ist und was sie abliefert. Die einzelnen Teile des Return-Pragmas haben die folgende Bedeutung:
StartSym ist das Startsymbol der Grammatik.
Spuren ist die Spurenliste des Startsymbols.
SpezFs ist die Liste der gesammelten speziellen Formen.
AnaErg ist das abzuliefernde Analyse-Ergebnis.
Rumpf ist Prolog-Code, der im Modul user ausgeführt wird.

Beispiel: pragma tomita_return(input, [], _, input:tree).
Die Analyse gelingt, wenn die Spurenliste leer ist. Sie liefert den Wert des Attributs input:tree. Die speziellen Funktionen werden nicht beachtet.

Beispiel: pragma tomita_return(s2, [], SpezFs, ZeigeTerm)
:- baueZeigeTerm(SpezFs, s2:tree, ZeigeTerm).
Die Analyse gelingt, wenn die Spurenliste leer ist. Beim Parsen wird die flache Semantik aufgebaut und zusammen mit der erzeugten Struktur des Merkmals s2:tree gezeigt.

pragma lazy_gen(Pfad)
wertet Pfad bei der Generierung verzögert aus. (s. Kapitel über Constraintaufspaltung bei der Generierung)

Beispiel einer Trace-Definition:
trace(_,np) | np:leer=yes.
Trace-Definitionen legen fest, welche Kategorie überhaupt bewegt werden können. Durch diese Definition kann jede np, die auf der rechten Seite einer Grammatikregelauftritt, zu leer expandieren. Um die Inflation der möglichen leeren Expansionen in der Grammatik zu begrenzen, führt man daher gewöhnlich bei der Trace-Definition ein Merkmal 'leer' ein, das im Falle der Bewegung zu 'yes' instantiiert ist. Soll eine np auf der rechten Seite einer Grammatikregel nicht bewegt werden können, intstantiiert man das Merkmal mit 'no'. Im Fall einer bewegbaren Konstituenten läßt man es uninstantiiert.

Beispiel eines Bounding-Nodes: bounding_node(np).
Bounding-Nodes sind Grenzkategorien bei der Bewegung von Konstituenten. Eine bewegte Konstituente vom Typ 'var' darf über einen, eine bewegte Konstituente vom Typ 'ana' darf über keinen Grenzknoten bewgt werden.

Beispiel einer Kopfdefinition:
v is_head_of vk.

Durch die Kopfdefinitionen wird vom Grammatikschreiber eine Linie für die Kopfbewegung festgelegt. Die Kopfbewegung dient im Deutschen für die Bewegung des finiten Verbs aus der Verberst- oder Verbzweitstellung in die Verbletztposition.

Beispiel einer Grammatikregel ohne Bewegung:
np - - -> np:np2 , np:np3 |
np2:morph:kas=np3:morph:kas.
Kommt eine Kategorie in einer Regel mehrfach vor, so müssen alle bis auf ein Vorkommen dieser Kategorie durch Indizes Kat:Index bezeichnet werden. Im Bedingungsteil wird dann der Index verwendet.

Beispiel einer Grammatikregel mit Bewegung:
s1 - - -> np<trace(var,np:npt) , s |
npt:morph=np:morph.
Die mit np<trace(var,np:npt) markierte Stelle definiert die Landeposition bei einer Argumentbewegung. In jeder weiteren Grammatikregel, die auf der rechten Seite eine np enthält, kann diese leer expandieren, wodurch sie zur Spurposition einer Bewegung wird. Ob sie tatsächlich auch in der angegebenen Position 'landen' kann, hängt von den zwischen den Merkmalsbeziehungen zwischen der Spurposition npt und der Landeposition np ab und davon, über wieviele Bounding-Nodes die Bewgung verläuft. Im Falle ...<trace(var,... darf die Bewegung über einen, im Falle ...<trace(ana,...jedoch über keinen Bounding-Node verlaufen. Man beachte auch, daß Spurposition und Landeposition nicht dieselbe Kategorie besitzen müssen.

Beispiel einer Grammatikregel für die Kopfbewegung:
s1 - - -> v + s |
Die links neben dem Symbol '+' auftretende Kategorie stellt die Landeposition für die Kopfbewegung dar. Bei der Kopfbewegung müssen im Gegensatz zur Argumentbewegung alle Merkmale zwischen Spur-und Landeposition identisch sein. Der Weg, der zwischen Spur- und Landeposition zurückgelegt wird, wird hierbei in der Grammatik durch die Angabe von is_head_of-Definitionen beschrieben.

Beispiele für Bedingungen:
Eine Disjunktion: np1:morph:pers='2' ; np1:morph:pers='3'
Eine Gleichung: np1:morph:agr:num=np2:morph:agr:num
Ein Makroaufruf: nominativ(np:morph:kas)
Eine Inklusion: np1:morph:pers in {'2','3'} (äquivalent zu obiger Disjunktion)
Ein Termaufbau: In der Regel np - - -> np:np2 , np:np3 kann durch die Verwendung von Backquote in der Gleichung np:tree=`np(np2:tree,np3:tree) auch ein neuer Term aufgebaut werden. So ergibt sich für das Merkmal np:tree der Wert, indem die Werte von np2:tree und np3:tree in die Struktur np(_,_) eingesetzt werden. Diese Art des Termaufbaus wird in den speziellen Formen insbesondere zum Aufbau der flachen Semantik eingesetzt.

Hinweis: Die hier angegebenen Erläuterungen stellen keine Dokumentation dar, sondern sind lediglich zur Illustration gedacht. Für eine ausführliche Beschreibung sei auf [Sch94],[BS92] und [Blo93] verwiesen.

4.3 Struktur eines Lexikons

Ein Lekikon im TrUG-Format besteht aus einem allgemeinen Deklarationsteil und den Lexikoneinträgen.

Lexikon → allgemeiner Deklarationsteil , Lexikoneintrag*

Ein Lexikoneintrag hat die Form:
lexicon(Wort,KatSym) | Bedingung* oder
lexicon(Wort,KatSym:Index) | Bedingung*

5 Die flache Semantik als Schnittstelle zwischen Analyse und Synthese

Die Schnittstelle zwischen Analyse und Synthese wird durch eine Datenstruktur gegeben, die das Format einer Liste hat und flache Semantik heißt.[2] Diese Namensgebung besagt, daß nicht kompliziert verschachtelten Baumstrukturen, sondern flache Strukturen als Zwischenrepräsentation aufgebaut werden, wobei der Schwerpunkt auf einer semantiknahen Repräsentation liegt. Das bedeutet aber nicht, daß in dieser Struktur ausschließlich semantische Information kodiert ist. Es handelt sich vielmehr eher um eine Strukturbeschreibungssprache, in der genausogut auch syntaktische Informationen beschrieben werden können. Zum Beispiel kann durch Prädikate auch die Konstituentenstruktur beschrieben werden.

5.1 Definition der flachen Semantik

Eine flache Semantik wird durch folgende Konstruktionsregeln beschrieben:[3]

flache Semantik → [semantischer Ausdruck$_1$, ... semantischer Ausdruck$_N$]
semantischer Ausdruck → semantische Gleichung | semantischer Term
semantische Gleichung → semantischer Term$_1$ = semantischer Term$_2$
semantischer Term → Atom | Compound-Prolog-Term

5.2 Beispiel einer flachen Semantik

In folgendem Beispiel sind neben der semantisch orientierten Prädikat-Argumentstruktur auch halb semantisch, halb syntaktische Eigenschaften (wie Person, Kasus, Genus, Numerus), Stellungseigenschaften (über Bewegung und Topikalisierung von

[2]Weitere Information über die flache Semantik befindet sich in [Hun93].

[3]Im Prinzip sind für Compound-Prolog-Terme beliebig tief geschachtelte Terme zulässig, doch läuft dies dem Gedanken an eine flache Struktur zuwider. Sinnvollerweise sollten daher zusammengesetzte Terme die Form f(x_1, ... x_n) haben, wobei die x_i Variablen oder Atome sind.

Argumenten und Adjunkten) sowie rein syntaktische Eigenschaften (Existenz von Verbargumentpositionen und kategoriale Information) ausgedrückt. Die einzelnen Prädikate sind dabei durch logische Variablen miteinander verbunden. Es besteht keine formale Trennung zwischen den verschiedenen linguistischen Wissensquellen. Die Auswahl und Anzahl der angegebenen Eigenschaften ist wegen der hochgradig modularen Repräsentation nicht a priori festgelegt, sondern jederzeit durch den Grammatikschreiber leicht veränderbar.

wo sollen wir uns treffen?

```
SEM =[
wh_locality(A,B),
perspron(C), num(C)=pl, pers(C)='1',gen(C)=D, kas(C)=kas(yes,no,yes),
perspron(E), num(E)=pl, pers(E)='1', gen(E)=F, kas(E)=kas(yes,yes,yes),
treffen(B,E,C), temp(B)=praes, mod(B)=ind, verb(B)=yes, ev_type(B)=event,
sollen(B), num(B)=pl, pers(B)='1',
bew(C)=yes, arg3(B,C),
bew(E)=yes, arg2(B,E), top(A)=yes,
adverbial(B,A)=top, prop_erst(B)=yes, satzmod(B)=frage
]
```

5.3 Die flache Semantik als Strukturbeschreibungssprache

An folgendem Beispiel der einfachen NP *der Termin* wird durch Numerieren der Knoten ersichtlich, wie die flache Semantik auch als Beschreibung der Konstituentenstruktur benutzt werden kann.

Konstituentenstruktur:
    np(det(der),n(Termin))
Konstituentenstruktur mit numerierten Knoten:
    np-1(det(der)-2,n(Termin)-3)
Beschreibung der Konstituentenstruktur in der flachen Semantik:
    knoten(1), cat(1)= np, string(1)=[der,Termin], sem(1)=[def(t), termin(t)], Töchter(1)=[2,3],
    knoten(2), cat(2)=det, string(2)=[der], sem(2)=[def(t)], Töchter(2)=[],
    knoten(3), cat(3)=n, string(3)=[Termin], sem(3)=[termin(t)], Töchter(3)=[]

5.4 Eigenschaften der flachen Semantik

Die erzeugte Repräsentation besitzt zwei formale Eigenschaften:

1. Sie ist permuationsinvariant bezüglich der auftretenden Prädikate.

2. Sie ist nicht invariant gegenüber Verdopplung von Prädikaten.

Die erste Eigenschaft ist dabei eine Folge der Interpretation der flachen Semantik als logische Konjunktion der auftretenden Prädikate. Die zweite Eigenschaft rührt daher, daß Wörter, die in einer Äußerung mehrfach vorkommen, in der flachen Semantik einen identischen Eintrag erzeugen können.

5.5 Initialisierungskommandos für die flache Semantik

Die flache Semantik wird initialisiert, wenn im Deklarationsteil für Grammatik und Lexikon die folgenden fünf Compileranweisungen aufgeführt sind:

```
pragma tomita_special(sem(term)).
pragma tomita_special(semdel(term)).
pragma tomita_special(semdelopt(term)).
pragma tomita_special(semzus(term)).
pragma tomita_return(⟨StartSym⟩, [], SpezFs, ZeigeTerm) :-
        baueZeigeTerm(SpezFs, ⟨StartSym⟩:tree, ZeigeTerm).
```

5.6 Aufbau der flachen Semantik im Lexikon

Ein Lexikoneintrag hat die Form
lexicon(Wort,KatSym) | Bedingung* oder
lexicon(Wort,KatSym:Index) | Bedingung*[4]

Ein Lexikoneintrag, der zunächst ja nur rein syntaktisches Wissen ausdrückt, kann im Bedingungsteil durch Einfügen semantischer Konstrukte einen Anknüpfungspunkt zur Semantikkonstruktion bzw. zur Generierung erhalten.

Hierdurch wird es möglich, jedes in der Beschreibung des Lexikoneintrages benutzte Merkmal in einer für den Grammatikschreiber transparenten und völlig modularen Weise nach Bedarf in die Semantik zu übernehmen. Der Grammatikschreiber wird dabei nur solche Merkmale aus der Beschreibung übernehmen, die er für die semantische Auswertung als relevant erachtet. Bei der praktischen Abwicklung von Projekten entsteht allerdings häufig das Problem, zu einer anfangs definierten Merkmalsmenge noch weitere Eigenschaften hinzuzufügen, die zu Beginn des Projektes noch nicht vorhanden waren oder deren Relevanz nicht erkannt wurde. Die flache Semantik unterstützt hier die Aufwärtskompatibilität bei der Grammatikentwicklung, sofern nur Merkmale hinzugefügt werden.

Die semantischen Konstrukte sind hierbei spezielle Ausdrücke, die mit den Funktoren 'sem', 'semdel' und 'semdelopt' gebildet sind. Genauer gilt:

[4] Es ist nicht erlaubt, für Kat eine Variable anzugeben, selbst wenn diese im nachfolgenden Bedingungsteil gebunden wird. Dies führt im günstigsten Fall zu einer Fehlermeldung, im ungünstigen Fall zur Nichtterminierung. (Fehler 1)

Semantikkonstrukt → direktes Semantikkonstrukt |
zusätzliches Semantikkonstrukt

direktes Semantikkonstrukt → sem(semantischer Term)
zusätzliches Semantikkonstrukt →
semdel(‘[semantischer Term_1, ... semantischer Term_N]) |
semdelopt(‘[semantische Gleichung_1, ... semantische Gleichung_N]) |
semzus(‘[semantischer Term_1, ... semantischer Term_N])
semantische Gleichung → semantischer Term_1 = semantischer Term_2
semantischer Term → Atom | Compound-Prolog-Term

Semantikkonstrukte dürfen dem Bedingungsteil nur konjunktiv mitgegeben werden. Der Generatorcompiler erlaubt dagegen nicht, daß in Disjunktionen Semantikkonstrukte auftreten.[5] (Fehler 2)

Semantische Terme im Lexikon

Ein Lexikoneintrag enthält im Bedingungsteil entweder einen einzelnen semantischen Term oder keinen semantischen Term(bei semantisch leeren Wörtern). Dieser Term darf keine Liste von Termen sein. (Fehler 3, Fehler 4, Fehler 5)

Ein zulässiger Eintrag wird durch (1) gegeben.

Beispiel: der Lexikoneintrag für *miteinschliessen* (1)

```
lexicon(Word , v) |
    sem(‘miteinschliessen(v:semsc:var , v:semsc:arg1 , v:semsc:arg3)) ,
    v:prt = [ein , mit] ,
    ⟨andere Bedingungen⟩.
```

Eine Besonderheit für den semantischen Funktor ergibt sich bei der Benutzung von Makros. Oben wurde verlangt, daß der semantische Funktor ein Atom ist. Nun kann man mit Hilfe von Makros sehr praktisch semantische Klassen bilden. Es tritt dabei das Problem auf, daß im Makro der semantische Funktor als Variable übergeben wird.

```
lexicon(Nomen,nr:n) |
    sem_noun(n,herr),
    ⟨weitere Bedingungen⟩.
```

Schreibt man nun das Makro als

[5] Dies scheint nicht erforderlich zu sein und hat überdies den Nachteil, daß selbst bei verzögerter Constraintauswertung Disjunktionen zurückbleiben, die sofort evaluiert werden müssen.

```
sem_noun(Cat,Wort) short_for
        sem(‘Wort(Cat:semsc)).
```

so tritt beim Einlesen ein Fehler auf, da Prolog keine variablen Funktoren unterstützt. Der Funktor könnte aber beim Evaluieren des Makros an ein Atom gebunden werden. Deswegen wird die Verwendung einer Gleichung =.. gestattet. Diese kann normal eingelesen und dann später evaluiert werden.

```
sem_kat1(W,Kat) short_for
        sem(_ =.. [W,Kat:var]),

lexicon(W,kat1) |
        sem_kat1(wort_grundform,kat1),
        ( W=wort1, kat1:feat1=wert1
        ;
          W=wort2, kat1:feat1=wert2
        ).
```

Die Nützlichkeit dieser Vorgehensweise zeigt sich insbesondere bei Lexikoneinträgen, bei denen die Semantik durch einen Featurewert als Funktor gesteuert wird wie im Eintrag für intransitive Präpositionen. Da für den Generator das Lexikon nach dem semantischen Funktor indiziert wird, entstehen aus diesem Eintrag dann 5 verschiedene Einträge im compilierten Generatorlexikon.

```
sem_pintr(Cat,Rel) short_for
        sem(_ =..[Rel,Cat:semsc]).

lexicon(W,pintr) |
        sem_pintr(pintr,W),
        pintr:sentarg=yes,
        ( W=darauf,
           pintr:rel=auf,
           pintr:tree=pintr(darauf)
        ;
           W=dabei,
           pintr:rel=bei,
           pintr:tree=pintr(dabei)
        ;
           W=daraus,
           pintr:rel=aus,
           pintr:tree=pintr(daraus)
        ;
           W=dafuer,
           pintr:rel=fuer,
           pintr:tree=pintr(dafuer)
        ;
```

```
    W=davon,
    pintr:rel=von,
    pintr:tree=pintr(davon)
  ).
```

Semantische Zusatzkonstrukte

Auch hier gibt es verschiedene Fehlerquellen für das Format der Zusatzkonstrukte, die in Fehler 6, Fehler 7 und Fehler 8 beschrieben sind.

Beispiel: der (vereinfachte) Lexikoneintrag für Termin (2)

```
lexicon(Nomen , n) |
    sem(`termin(n:semsc:var)) ,
    semdelopt(`[num(n:semsc:var)=n:morph:agr:num ,
                kas(n:semsc:var)=n:morph:kas]) ,
    n:morph:gen=mask ,
    ( Nomen = 'Termin' ,
      n:morph:agr:num=sg ,
      n:morph:kas:notgen=yes ,
      n:tree = n('Termin')
    ;
      Nomen = 'Termins' ,
      n:morph:agr:num=sg ,
      n:morph:kas:notgen=no ,
      n:morph:kas:notnom=yes ,
      n:morph:kas:dir=no ,
      n:tree = n('Termins')
    ;
      Nomen = 'Termine' ,
      n:morph:agr:num=pl ,
      n:morph:kas:notgen=n:morph:kas:dir ,
      n:tree = n('Termine')
    ;
      Nomen = 'Terminen' ,
      n:morph:agr:num=pl ,
      n:morph:kas:notgen=yes ,
      n:morph:kas:notnom=yes ,
      n:morph:kas:dir=no ,
      n:tree = n('Terminen')
    ).
```

Aufbau der flachen Semantik beim Parsen

Beim Parsen einer Äusserung werden die in semdelopt(⟨Liste⟩), semdel(⟨Liste⟩) und semzus(⟨Liste⟩) befindlichen Prädikate mit dem in sem(⟨Term⟩) enthaltenen Term zu einer Liste konkateniert. Beispiel: Beim Parsen von *Termins* entsteht die flache Semantik [termin(B), num(B)=sg, kas(B)=kas(no,yes,no)].

Einschränkung der Generierungen durch die Zusatzkonstrukte
Die in den semantischen Zusatzkonstrukten beschriebenen Eigenschaften stellen Beschränkungen für die erzeugten Generierungen dar. Der Generator greift dabei zunächst mit Hilfe der als Schlüssel verwendeten Wortsemantik auf das Lexikon zu, wobei im Bedingungsteil noch ungelöste Bedingungen enthalten sind. semzus-Ausdrücke erzeugen dabei keine Einschränkung der Generierung.

Einschränkung der Generierungen durch semdelopt-Ausdrücke
Annahme 1: Der Lexikoneintrag enthält in semdelopt die Gleichung Func=Val.
Annahme 2: Die Eingabesemantik beim Generieren enthält die Gleichung Func'=Val'.
Dann gilt: Wenn durch das Lösen des Bedingungsteils des Lexikoneintrages Func ein ground-Term ist, der mit Func' unifiziert, so muß Val=Val' gelten.

Damit optionale Einschränkungen überhaupt wirksam werden können, müssen daher die logischen Variablen in der Semantik des Wortes instantiiert sein.

Eingabesemantik = [termin(B), num(t)=sg, kas(t)=kas(no,yes,no)]
Generierungen: *Termin, Termins, Termine, Terminen*

Durch Lösen des Bedingungsteils des Lexikoneintrages wird in num(n:semsc:var) = n:morph:agr:num das Merkmal n:semsc:var mit der Variablen B belegt, ist somit nicht ground. Es erfolgt daher keine Einschränkung der Generierung.

Eingabesemantik = [termin(t), num(_)=sg, kas(_)=kas(no,yes,no)]
Generierungen: *Termins*

Durch Lösen des Bedingungsteils des Lexikoneintrages wird in num(n:semsc:var) = n:morph:agr:num das Merkmal n:semsc:var mit dem Atom t belegt. Unifikation mit num(_) ist möglich. Es gilt daher sg = n:morph:agr:num. Ebenso ist Unifikation mit kas(_) möglich. Es gilt daher auch kas(no,yes,no) = n:morph:kas. Im allgemeinen sollten die logischen Variablen der Prädikate der Eingabesemantik durch Atome instantiiert sein, da es sonst zu durchaus nicht beabsichtigten Variablenbindungen kommen kann. Man sollte also besser die Eingabesemantik [termin(t), num(t)=sg, kas(t)=kas(no,yes,no)] anstelle von [termin(t), num(_)=sg, kas(_)=kas(no,yes,no)] wählen.

Einschränkung der Generierungen durch semdel-Ausdrücke
Annahme : Der Lexikoneintrag enthält in semdel das Prädikat P.
Dann muß die Eingabesemantik ein mit P unifizierbares Prädikat P' enthalten, wenn die Generierung gelingen soll. Für jedes derartige Prädikat P' der Eingabesemantik

werden alle zulässigen Variablenbelegungen bestimmt, wodurch im Bedingungsteil ein zusätzlicher Constraint entsteht.

Beispiel: Zerlegung eines Kompositums in mehrere Prädikate.
Damit das Kompositum generiert werden kann, müssen alle Prädikate in der Eingabesemantik definiert sein. Auf diese Weise können einem Wort auch mehrere Prädikate zugeordnet werden. Der Grammatikschreiber muß lediglich eines dieser Prädikate als Schlüssel für den Lexikonzugriff angeben.

```
lexicon(Nomen , n) |
        sem(‘essen(n:semsc:var)) ,
        semdel(‘[am(_X,n:semsc:var),abend(X)]) ,
        ⟨Wortdisjunktionen und andere Bedingungen für Abendessen⟩.
```

5.7 Aufbau der flachen Semantik in der Grammatik

Das Kompositionalitätsprinzip

Für jede Grammatikregel ⟨Linke Seite⟩ → ⟨rechte Seite⟩ | ⟨Bedingungen⟩ , die im Bedingungsteil keine semantischen Zusatzterme enthält, gilt das Kompositionalitätsprinzip. Die flache Semantik der linken Seite ist hierbei gleich der Vereinigung(= ListenKonkatenation) der flachen Semantiken der Kategorien auf der rechten Seite. Hierbei werden alle durch sem, semdel, semdelopt und semzus definierten Terme in einer Liste vereinigt.

Ein Beispiel für eine strikt kompositionelle Regel wird durch die Regel gegeben, bei der das Dativobjekt an das Verb gebunden wird.

Sprachlicher Ausdruck: *dem Sekretariat schickt*

```
vp → np, v1 |
        dat_or_akk(np:morph),
        vp:semsc:arg2=np:semsc:var,
        vp:semsc=v1:semsc,
        ⟨andere Bedingungen⟩.
```

Flache Semantik: [def(s),sekretariat(s),schicken(Ev,A,s,C)]

Semantische Monotonie

Das Kompositionalitätsprinzip verliert seine Gültigkeit und muß durch das schwächere Prinzip der semantischen Monotonie ersetzt werden, falls eine Regel einen semdel(⟨Liste⟩)-Ausdruck erhält. Die durch einen derartigen Ausdruck in die Semantik eingefügten Prädikate sind nämlich kein Bestandteil der flachen Semantiken der Kategorien auf der rechten Seite der Regel, sondern werden explizit durch die Regel eingeführt. Immerhin wird die bei Gültigkeit des Monotonieprinzips die Semantik noch additiv aufgebaut aus den Semantiken der Töchter und den für die Regel maßgeblichen Zusatztermen.

Monotonieprinzip Die Semantik einer Kategorie auf der rechten Regelseite subsumiert die Semantik der linken Regelseite.

Ein Beispiel ist die Regel für die Bildung des Genitivattributs, dessen Semantik insofern mehr als die Vereinigung der Semantiken der Kategorien auf der rechten Regelseite ist, als eine zusätzliche Besitzrelation eingeführt wird.

Sprachlicher Ausdruck: *Peters Termin*
np $\rightarrow$ np^{gen} , np^2 |
    semdel([poss(np^{gen}:var,np^2:var)]),
    $\langle$andere Bedingungen$\rangle$.
Flache Semantik: [peter(p),termin(t),poss(p,t)]

Einschränkung der Generierungen durch die Zusatzkonstrukte
Jedem Lexikoneintrag entspricht genau ein semantischer Kernterm, der durch dessen sem-Ausdruck erzeugt wird. Diese Kernterme sind für die Generierung erforderlich, da mit ihrer Hilfe der Zugriff auf das Lexikon erfolgt. Dabei ist die syntaktische Kategorie zu einem Kernterm nicht notwendigerweise eindeutig bestimmt. So kann z.B. dem Prädikat fahren(Ev,X) das Verb *fahren* oder das Nomen *Fahrt* zugeordnet werden. Beim Generieren wird die Semantik zunächst in zwei Mengen M1 und M2 aufgespalten. Jeder Term in M1 entspricht dabei einem Lexikoneintrag. In M2 entspricht kein Term einem Lexikoneintrag. Zu den Termen aus M1 werden sodann Lexikoneinträge unter Berücksichtigung der durch M2 wirksamen Beschränkungen gefunden. Diese werden durch die Anwendung von Grammatikregeln stückweise nach dem CKY-Algorithmus solange zusammengesetzt, bis die Eingabesemantik aufgebraucht ist. Der Mechanismus der Einschränkung der Generierungen durch die Zusatzterme ist dabei derselbe wie im Lexikon. Bezüglich der Anforderungen an die semantische Eingangsstruktur für den Generator im Hinblick auf die Charakterisierung der semantischen Terme ergibt sich folgendes Bild:

semzus-Terme	für die Generierung irrelevant
sem-Terme	für die Generierung erforderlich
semdel-Terme	für die Generierung erforderlich
semdelopt-Terme	für die Generierung nicht erforderlich, aber einschränkend

5.8 Motivation der flachen Semantik als Syntax-Semantik-Schnittstelle

Für die Auswahl der flachen Semantik als Repräsentationsebene zwischen Syntax und Semantik sprechen folgende Gründe:

1. Gewährleistung der Reversibilität
2. Einfachheit des Formalismus
3. Ökonomie der Grammatikentwicklung

4. maximale Modularität

5. leichte Interpretierbarkeit

6. hohe Flexibilität und Wartbarkeit

ad 1) Die flache Semantik ist einerseits eine Repräsentation, in die leicht abgebildet werden kann. Andererseits kann aus dieser Repräsentation Sprache auch wieder erzeugt werden.

ad 2) Die flache Semantik bietet einen sehr einfachen Formalismus an, der auch von Nichtexperten leicht verstehbar ist. Dies macht ihre Anwendung auch für potentielle industrielle Anwender interessant, die gegebene Inhalte als Konjunktion von Prädikaten repräsentieren und durch die Generierung dann eine sprachliche Darstellung dieser Inhalte erhalten können.

ad 3) Die Entwicklung von Grammatiken, die über eine breite Abdeckung der sprachlichen Phänomene verfügen, beansprucht einen langen Zeitraum. Bisher wurden als semantische Zwischenrepräsentation komplexe Termstrukturen benutzt, welche gesonderte Interpretationsregeln für den Aufbau der semantischen Struktur erforderlich machten. Dieser Schritt wird nun wesentlich vereinfacht, da die Semantik additiv durch Vereinigung der Semantiken der Töchter und der Zusatzterme erzeugt wird.

ad 4) Zur Modularität ist anzumerken, daß die Sprachanalyse Informationen erzeugt, die von nachfolgenden Verarbeitungskomponenten benötigt werden. Durch die inhaltliche Neutralität der flachen Semantik kann jede gewünschte Information verfügbar gemacht werden, wobei der Zugriff auf die benötigten Wissenseinheiten regelbasiert erfolgt. Ein 'Zuviel' an Information stört dabei nicht die nachfolgenden Verarbeitungskomponenten, da sich diese die für sie relevanten Informationen aus dem Pool von Informationen, den die flache Semantik darstellt, selbständig heraussuchen. Insbesondere wird die Generierung a priori nicht davon berührt, ob noch zusätzlich eine Konstituenten-, Diskursrepräsentations-, Quantoren- oder sonstige Struktur dargestellt wird. Die flache Semantik ist daher mit dem Konzept der variablen Analysetiefe verträglich.

ad 5) Die Benutzung der flachen Semantik als Repräsentationssprache macht es möglich, beliebige Strukturen zu beschreiben, die durch nachfolgende Interpreter dann tatsächlich aufgebaut werden können. Zum Beispiel kann die für die Quantorenstruktur wichtige Oberflächenreihenfolge repräsentiert werden, ohne daß diese während der syntaktischen Analyse schon direkt aufgebaut werden muß. Die Feststellung der Quantifikationseigenschaften ist nämlich von vielen anderen Faktoren abhängig, wobei die Oberflächenreihenfolge zwar eine wichtige Rolle spielt, die Quantorenstruktur aber nicht vollständig determiniert. Auf diese Weise können verschiedene Verarbeitungskomponenten auf einer Struktur arbeiten. Wichtig ist dabei, daß das Wissen aus früheren Verarbeitungsschritten erhalten bleibt und nicht in Vergessenheit gerät. Wird dagegen eine Quantorenstruktur als Zwischenrepräsentation gewählt, so ist von der Oberflächenreihenfolge abstrahiert worden, so daß diese bei der Generierung nicht mehr beeinflußt werden kann. Außerdem müssen dann auf der Analyseseite entweder mehrere alternative Strukturen oder eine unterpezifizierte Struktur angegeben werden, um die Anzahl der möglichen

Quantorenstrukturen zu beschreiben. Die Wahl der flachen Semantik als Repräsentation verzichtet demgegenüber auf die direkte Interpretation und stellt nur die von der einen Verarbeitungskomponente gelieferten Wissenseinheiten so zur Verfügung, daß sie die Arbeit der anderen Verarbeitungskomponenten ermöglicht, aber nicht notwendigerweise erzwingt.

ad 6) Durch die flache Art der Darstellung läßt sich die Struktur sehr einfach durch Einführung neuer Prädikate erweitern, ohne daß die Zugriffsfunktionen für bereits existierende Informationen geändert werden müssen. Der Zugriff auf Informationen in einer stark geschachtelten Termstruktur ist demgegenüber weit weniger flexibel durchführbar, da bei jeder Erweiterung der Struktur die Zugriffsfunktionen geändert werden müssen. Dies erübrigt sich bei der flachen Semantik infolge der Permutationsinvarianz der Darstellung und des regelbasierten Zugriffs.

Als Beispiel eines Interpreters für flache Strukturen kann der nachstehend beschriebene Transferlexikon-Compiler betrachtet werden.

6 Der Compiler für das Transferlexikon

6.1 Aufruf des Compilers für das Transferlexikons

ttlc TransferLexikondatei
übersetzt die TransferLexikondatei ⟨TransferLexikondatei⟩ in die QOF-Datei ⟨TransferLexikondatei⟩.qof und die Textdatei ⟨TransferLexikondatei⟩.pl

6.2 Motivation

Präpositionen besitzen üblicherweise eine Semantik z.B. insbesondere dann, wenn sie eine eine lokale oder temporale Angabe bezeichnen.

Sprachlicher Ausdruck: *auf dem Platz*
Flache Semantik: [auf(a,p,_), def(p), Platz(p)]

Sprachlicher Ausdruck: *auf dem Platz warten*
Flache Semantik: [auf(a,p,ev), def(p), Platz(p),warten(ev,_,_)]

In Kontexten, wo die Präposition jedoch zur Bildung eines Präpositionalobjektes gebraucht wird, verliert die Präposition ihre Semantik und wird zum reinen Kasuszuweiser. Sie sollte daher nicht in der semantischen Form erscheinen. Man erwartet daher eine Semantik wie in (a) und nicht wie in (b).

(3)

Ich warte auf Sie.
(a1) [warten(ev,i,s), perspron(i), num(i)=sg, pers(i)=1,
perspron(s), num(s)=pl, pers(s)=3]
(b1) [warten(ev,i,s), perspron(i), num(i)=sg, pers(i)=1,
perspron(s), num(s)=pl, pers(s)=3, auf(as,s,ev)]

Ich warte auf Sie auf dem Platz.
(a2) [warten(ev,i,s), perspron(i), num(i)=sg, pers(i)=1,
perspron(s), num(s)=pl, pers(s)=3,
auf(a,p,ev), def(p), Platz(p)]
(b2) [warten(ev,i,s), perspron(i), num(i)=sg, pers(i)=1,
perspron(s), num(s)=pl, pers(s)=3, auf(as,s,ev),
auf(a,p,ev), def(p), Platz(p)]

Ähnliche Überlegungen gelten für inhärent reflexive Verben und das unpersönliche *es*.

Allen diesen Fällen ist gemeinsam, daß es sich um lexikalisierbare Eigenschaften der verwendeten Wörter handelt. Da die Semantik additiv aufgebaut wird und beim Parsen von (3) zunächst die Form (b) entsteht, muß der Term auf(a,s,ev) aus (b) gelöscht werden, um (a) zu erhalten.

Dies erfolgt mittels einer Transferregel

[warten(Ev,A1,I)] '⟨-t-⟩' [warten(Ev,A1,Inh),auf(Inh,I,Ev)]. (4)

durch welche die grammatiknähere Darstellung auf der rechten in die anwendungsnähere Darstellung auf der linken Seite transformiert wird. Zum Generieren muß ja die Präposition *auf* erzeugt werden, ohne daß sie explizit in der Eingabestruktur für den Generator spezifiziert werden soll. Vielmehr handelt es sich bei der Auswahl einer geeigneten Präposition um eine lexikalische Eigenschaft des verwendeten Verbs. Diese erfolgt durch die Festlegung eines geeigneten Transferlexikons, in dem die assoziierten Prädikate regelweise über ihre logischen Variablen miteinander verknüpft werden.

Bei der Systemausgabe der semantischen Struktur des Parsers wird an der Oberfläche nur das anwendungsnähere Format sichtbar, wogegen die grammatiknähere Darstellung als internes Format betrachtet wird. Die Eingabestruktur für den Generator wird durch das anwendungsnähere Format gegeben, welches beim Generieren vermittels der Transferregel (4) automatisch das grammatiknähere Format erzeugt.

6.3 Format einer Transferregel

Eine Transferregel hat das Format ⟨Linke Seite⟩ ⟨Operator⟩ ⟨Rechte Seite⟩.
⟨Operator⟩ ist einer der drei Operatoren '-t-⟩' ,'⟨-t-' , '⟨-t-⟩'.
⟨Linke Seite⟩ und ⟨Rechte Seite⟩ sind jeweils Listen von Termen.

6.4 Format einer Transferlexikons

Ein Transferlexikon hat die Struktur

1. Eine Startklausel transfer(⟨Sprache1⟩, ⟨Sprache2⟩)
2. endlich viele Regeln ⟨Linke Seite⟩ ⟨Operator⟩ ⟨Rechte Seite⟩
3. eine optionale Defaultklausel default_clause(⟨Sprache1⟩, ⟨Sprache2⟩)

Die Startklausel gibt an, zwischen welchen Sprachen der lexikalische Transfer durchgeführt werden soll. Es können daher mehrere Transferlexikas gleichzeitig geladen sein. Falls keine der in 2. angegebenen Regeln ausgeführt werden kann, wird bei vorhandener Defaultklausel die Struktur als transferinvariant betrachtet. Ist keine Defaultklausel vorhanden, schlägt der Transfer fehl. Für alle Regeln wird die Eingangsstruktur nämlich nach dem Muster auf der linken Seite durchsucht und dieses durch das Muster auf der rechten Seite ersetzt.

6.5 Aufruf des Transferprädikates

transfer(⟨Sprache1⟩_⟨Sprache2⟩, ⟨Eingabe⟩, ⟨Ausgabe⟩) bzw.
transfer(⟨Sprache2⟩_⟨Sprache1⟩, ⟨Eingabe⟩, ⟨Ausgabe⟩) sind Prologprädikate, welche den Transfer ausführen. Diese Prädikate sind das Ergebnis der Compilation des Transferlexikons. Für die speziellen Sprachen ⟨Sprache1⟩=parse, ⟨Sprache2⟩=generate wird der Transfer automatisch ausgeführt, ist als gewissenmaßen built-in. Für andere Sprachen muß das betreffende Prologprädikat explizit aufgerufen werden.

6.6 Beispiele für Transferlexikas

Beispiel 1: Vermittlung zwischen dem anwendungsnäheren Format, wie es der Parser abliefert und dem grammatiknäheren Format, welches der Generator benötigt.

```
transfer(parse,generate).
[warten_auf(Ev,A1,I)] '<-t->' [warten_auf(Ev,A1,Inh),auf(Inh,I,Ev)].
[sich_aneignen(Ev,A1,A2)] '<-t->'
  [sich_aneignen(Ev,A1,A2),perspron(Ev)].
[sich_wenden_an(Ev,A1,I)] '<-t->'
  [sich_wenden_an(Ev,A1,Inh),perspron(Ev),an(Inh,I,Ev)].
es_belassen_bei(Ev,A1,I)] '<-t->'
  [es_belassen_bei(Ev,A1,Inh),perspron(Ev),bei(Inh,I,Ev)].
[es_sich_handeln_um(Ev,Inh)] '<-t->'
  [es_sich_handeln_um(Ev,I),perspron(Ev),perspron(Ev),
  pers(Ev)='3',num(Ev)=sg,gen(Ev)=neutr,um(Inh,I,Ev)].
default_clause(parse,generate).
```

Das allgemeine Schema dieser Regeln besteht darin, daß auf der semantiknähereren Seite nur ein Prädikat vorliegt, welches aber durch mehrere Wörter realisiert

werden muß, wobei diese zusätzlichen Wörter keinen eigenen Beitrag zur Semantik leisten, aber aus syntaktischen Gründen erforderlich sind. So sind bei dem Prädikat es_sich_handeln_um die Präposition *um* zur Bildung des Präpositionalobjektes, das unpersönliche *es* und das inhärente Reflexivum *sich* erforderlich, um den korrekten deutschen Satz *Es handelt sich um ein Treffen.*
bilden zu können.

Beispiel 2: Löschung von nicht zur Generierung erforderlicher Information. Nach dem Parsen enthält die Semantik Informationen über Kasus,Topic,bewegte Konstituenten, Satzmodus, Ereignistyp, Genus, Person u.a., die zum Generieren nicht erforderlich sind. Diese können mittels der folgenden Regeln entfernt werden.

```
transfer(max_spez,min_spez).
[kas(_)=_] '-t-⟩' [].
[top(_)=_] '-t-⟩' [].
[bew(_)=_] '-t-⟩' [].
[ev_type(_)=_] '-t-⟩' [].
[satzmod(_)=_] '-t-⟩' [].
[gen(_)=_] '-t-⟩' [].
[pers(_)=_] '-t-⟩' [].
default_clause(max_spez,min_spez).
```

7 Das Ablaufsystem für die Analyse

7.1 Laden der Grammatik

ana_gram_lade(+Grammatik)
lädt die in ⟨Grammatik⟩ enthaltene Grammatik in das TrUG-Ablaufsystem
und aktiviert sie für die Analyse. ⟨Grammatik⟩ muß bereits mit tagc übersetzt worden sein.

7.2 Laden des Lexikons

ana_lex_neu
teilt dem Trug-Ablaufsystem mit, daß ein neues Lexikon für die Analyse geladen wird.

ana_lex_lade(+Lexikonteile)
lädt die in ⟨Lexikonteile⟩ enthaltenen Teile des Lexikons in das Trug-Ablaufsystem
und aktiviert sie für die Analyse. ⟨Lexikonteile⟩ ist der Name einer Lexikondatei oder eine Liste von solchen Namen. Die Lexikondateien müssen bereits mit talc übersetzt worden sein.

7.3 Laden des Transferlexikons

transfer_lex_lade(+Lexikon)
lädt die Transferlexikondatei ⟨Lexikon⟩ in das Trug-Ablaufsystem.
Die Transferlexikondatei muß mit ttlc übersetzt worden sein.

7.4 Funktionen für den Ablauf der Analyse

parser.
startet eine Schleife, die jedesmal eine Zeile vom momentanen Eingabestrom liest, analysiert und alle von der Analyse zugeordneten Ergebnisse in den momentanen Ausgabestrom druckt. Die Eingabe von CTRL-D stoppt die Schleife. Ein Ergebnis ist ein Paar, bestehend aus einem Baum und einer Semantik.

parser(*AnaErg)
liest eine Zeile vom momentanen Eingabestrom und analysiert sie.
⟨AnaErg⟩ ist ein von der Analyse zugeordnetes Ergebnis.
Durch Rücksetzen erhält man alle Möglichkeiten.

parse(+ASCIIs)
analysiert ⟨ASCIIs⟩ und druckt alle von der Analyse zugeordneten Ergebnisse in den momentanen Ausgabestrom.

parse(+ASCIIs,*AnaErg)
analysiert ⟨ASCIIs⟩. ⟨AnaErg⟩ ist ein von der Analyse zugeordnetes Ergebnis.
Durch Rücksetzen erhält man alle Möglichkeiten.

8 Das Ablaufsystem für die Synthese

8.1 Laden der Grammatik

syn_gram_lade(+Grammatik)
lädt die in ⟨Grammatik⟩ enthaltene Grammatik in das TrUG-Ablaufsystem und aktiviert sie für die Analyse. ⟨Grammatik⟩ muß bereits mit tsgc übersetzt worden sein.

8.2 Laden des Lexikons

syn_lex_neu
teilt dem Trug-Ablaufsystem mit, daß ein neues Lexikon für die Synthese geladen wird.

syn_lex_lade(+Lexikonteile)
lädt die in ⟨Lexikonteile⟩ enthaltenen Teile des Lexikons in das Trug-Ablaufsystem und aktiviert sie für die Synthese. ⟨Lexikonteile⟩ ist der Name einer Lexikondatei oder eine Liste von solchen Namen.
Die Lexikondateien müssen bereits mit tslc übersetzt worden sein.

8.3 Laden des Transferlexikons

transfer_lex_lade(+Lexikon)
lädt die Transferlexikondatei ⟨Lexikon⟩ in das Trug-Ablaufsystem.
Die Transferlexikondatei muß mit ttlc übersetzt worden sein.

8.4 Funktionen für den Ablauf der Synthese

generiere(+Verbos,?Kat,+*SemI)
berechnet aus der Eingabesemantik ⟨SemI⟩ und druckt das Ergebnis in den momentanen Ausgabestrom.
Ist die syntaktische Kategorie ⟨Kat⟩ gesetzt, so werden nur Generierungen zu dieser Kategorie ausgegeben.
Das Atom Verbos steuert die Art des Ausdruckens. Dabei bestehen folgende Möglichkeiten:
nicht verboses Ausdrucken:
(nur die erzeugten Strings werden ausgegeben):
s schrittweises Ausdrucken pro Subsubzustand
n schrittweises Ausdrucken pro Subzustand
a komplettes Ausdrucken

verboses Ausdrucken:
(Zusatzinformation über die erzeugten Zuständen und die benötigten Zeiten wird ausgegeben)
sv schrittweises Ausdrucken pro Subsubzustand
nv schrittweises Ausdrucken pro Subzustand
av komplettes Ausdrucken

Für das nicht verbose Ausdrucken mit zusätzlicher Angabe der Generierungszeit wählt man:
sg schrittweises Ausdrucken pro Subsubzustand
ng schrittweises Ausdrucken pro Subzustand
ag komplettes Ausdrucken

Soll das Ausdrucken auf Strings maximaler Länge beschränkt werden, wählt man
sm schrittweises Ausdrucken pro Subsubzustand, nicht verbos
nm schrittweises Ausdrucken pro Subzustand, nicht verbos
am komplettes Ausdrucken, nicht verbos
svm schrittweises Ausdrucken pro Subsubzustand, verbos
nvm schrittweises Ausdrucken pro Subzustand, verbos
avm komplettes Ausdrucken, verbos
sgm schrittweises Ausdrucken pro Subsubzustand, nicht verbos + Generierungszeit
ngm schrittweises Ausdrucken pro Subzustand, nicht verbos + Generierungszeit
agm komplettes Ausdrucken, nicht verbos + Generierungszeit

Bei schrittweisem Ausdrucken erhält man die jeweils nächste Lösung durch Rücksetzen. Nur für Inspektionszwecke wird man den verbosen Modus wählen. Für die Erstellung eines Testfiles von Generierungen eignet sich am besten der nicht verbose Modus mit der Option sgm(Ausgabe der ersten maximalen Generierung) bzw. agm(Ausgabe aller maximalen Generierungen)

gen_to_file(+Datei,+*Kategorie)
liest die Datei ⟨Datei⟩ von Eingabetermen
und wertet diese aus.

Erzeugt durch Ausdrucken der Ergebnisse die Datei ⟨Datei⟩.gens

8.5 Inspektion von Generatorzuständen

Die während der Generierung erzeugten Generatorzustände werden zum Zwecke der Inspektion in Form einer Tabelle in die Datenbasis assertiert, falls der verbose Modus gewählt wurde. Die Tabelle wird durch das Prädikat gen_arr_zst(N, Sem, Cat, SubSubZst) gegeben. Dabei ist ⟨N⟩ die Länge des erzeugten Strings, ⟨Sem⟩ die zur Erzeugung des Strings verbrauchte Semantik, ⟨Cat⟩ die syntaktische Kategorie und ⟨SubSubZst⟩ eine Liste von Unterzuständen. Da die Semantik aus Effizienzgründen in eine bitset-Menge kodiert ist und ⟨SubSubZst⟩ groß sein kein, ist der direkte Zugriff auf diese Tabelle nicht ratsam. Es werden geeignete Zugriffsfunktionen zur Verfügung gestellt, bei der ⟨Sem⟩ in dem für den Anwender sichtbaren Listenformat kodiert ist. Jeder Unterzustand besteht seinerseits aus einer Schablone(=Merkmalsstruktur), einem direkt evaluierten Constraint, einem verzögert ausgewerteten Constraint und einem String.

get_gen_zst(+*N,+*Sem,+*Cat)
greift auf einen Generatorzustand der Länge ⟨N⟩, der Semantik ⟨Sem⟩ und der Kategorie ⟨Cat⟩ zu.
⟨Sem⟩ wird dabei im Listenformat angegeben.
Der Ausdruck der Unterzustände erfolgt mittels Rücksetzen. Die Constraints werden ungelöst ausgedruckt.

get_gen_zst_sol(+*N,+*Sem,+*Cat)
ruft get_gen_zst(N,Sem,Cat) auf, löst die Constraints und bestimmt zu jeder Lösung den zugehörigen String.

max_sol(+*Cat)
bestimmt die maximale Stringlänge Max und ruft get_gen_zst_sol(Max,_,Cat).

Auf diese Weise kann man sich z.B. gezielt alle Satzgenerierungen ausgeben lassen, bei der die Semantik komplett verbraucht wurde.

stat_subzste
druckt eine Statistik über die erzeugten Zustände aus.
Für jede Länge und jede Kategorie werden bestimmt:
- die Anzahl von Zuständen dieser Kategorie
  (Zahlen grösser 1 ergeben sich, wenn Zustände unterschiedlicher
  Semantik, aber gleicher Kategorie existieren.)
- die Gesamtanzahl von Unterzuständen

Hierdurch erhält man einen groben Überblick über den bei der Generierung auftretenden Indeterminismus.

print_lex
druckt die im Lexikon vorkommenden Semantiken mit den zugehörigen lexikalischen Kategorien und ihren Vielfachheiten aus.

lex_zst(+*Sem,+*Subzustände)
druckt zur Wortsemantik ⟨Sem⟩ die Subzustände durch Rücksetzen in den momentanen Ausgabestrom.

9 Semantisch leere Wörter

9.1 Motivation

Ohne semantisch leere Wörter ist zur Generierung jedes Wortes ein Eingabeprädikat für dieses Wort erforderlich.Aus der Sichtweise der Semantik ist dies ein unerwünschter Effekt, da auch Wörter, die zur Semantik der Äußerung nichts beitragen, durch ein Prädikat beschrieben werden müssen. In (5) erwartet man eine Modellierung der Prädikatmenge etwa der folgenden Art:

Karl verspricht zu kommen. (5)
[karl(k), versprechen(ev,k,ev2), kommen(ev2,k)]

Hierbei wird dem Infinitiv-zu kein semantisches Prädikat zugeordnet. Ein anderes Beispiel wird durch die Verben mit abtrennbarem Verbpartikel gegeben.

Schließt der Termin den Montag mit ein? (6)
[miteinschliessen(ev,p,m), def(p), termin(p), montag(m), def(m)]

Hier entsprechen dem einen Prädikat miteinschliessen(ev,p,v) drei Wörter, wogegen dasselbe Prädikat in der Nebensatzkonstruktion durch ein einziges Wort realisiert wird:
... weil der Termin den Montag miteinschließt.
Nimmt man nun an, daß den Verbpartikeln *ein,mit* kein eigenes Prädikat zugeordnet wird, sie also semantisch leer sind, so kann die während des Generierungsprozesses mittels der grammatischen Beschreibung automatisch erfolgte Aufspaltung in drei Wörter und deren richtige Positionierung hierdurch modelliert werden.

9.2 Ein Beispiel

Ein Lexikoneintrag hat das Format lexicon(Wort,Kat) | Bedingung. Enthält der Bedingungsteil keinen semantischen Term für die Wortsemantik, so ist das Wort semantisch leer. Dies ist bei den Verbpartikeln in (7) der Fall.

```
lexicon(Word,vprt) |                                   (7)
   vprt:rel=Word,
   ( Word=ab
   ;
       ....
   ;
    Word=ein
   ;
       ....
   ;
    Word=mit
   ;
       ....
    Word=zu
   ).
```

Der Lexikoneintrag für *miteinschließen* s.(8) enthält dagegen einen semantischen Term

```
lexicon(Word , v) |                                                        (8)
       sem('miteinschliessen(v:semsc:var , v:semsc:arg1 , v:semsc:arg3)) ,
       v:prt = [ein , mit] ,
       ⟨andere Bedingungen⟩.
```

9.3 Einbeziehung der semantisch leeren Wörter

Semantisch leere Wörter sind Wörter, denen im Lexikon kein Ausdruck sem(term) zugeordnet wurde. Diese Wörter werden in einer eigenen Datei zusammengefaßt. Das Lexikon besteht daher aus zwei Dateien, die mit Hilfe des include-Pragmas zu einer Datei zusammengefaßt werden. Ferner sind die semantisch leeren Wörter zusätzlich in der Grammatik noch einmal aufzuführen. Das allgemeine Schema kann dann wie folgt beschrieben werden:

Das Lexikon besteht daher aus zwei(oder mehr) Dateien.
 Lexikondatei1 (das sind die semantisch nichtleeren Wörter)
 Lexikondatei2 (das sind die semantisch leeren Wörter)

Aufbau des Gesamtlexikons:
 Lexikondatei enthält
 pragma include Lexikondatei1
 pragma include Lexikondatei2

Aufbau der Grammatik:

Grammatikdatei enthält
pragma include Lexikondatei2
andere Einträge

Die Compiler für Grammatik und Lexikon wirken dabei in verschiedener Weise auf die semantisch leeren Wörter.

Der Parsercompiler für das Lexikon berücksichtigt alle Wörter. Semantisch leere Wörter erzeugen beim Parsen keinen Eintrag in der Semantik.

Der Parsercompiler für die Grammatik überliest dabei die leeren Wörter in der Grammatik.

Der Generatorcompiler für das Lexikon überliest dabei die leeren Wörter aus dem Lexikon unter Ausgabe von Warnungen Die Warnungen dienen zur Kontrolle. Der Benutzer wird darüber informiert, welche Wörter keine Semantik haben.

Der Generatorcompiler für die Grammatik Die semantisch leeren Wörter werden als leere Expansionen in der Grammatik aufgefaßt und erzeugen beim Compilieren eine neue Grammatik, in der die leeren Expansionen eliminiert sind.

9.4 Probleme mit semantisch leeren Wörtern

Für den Parser sind semantisch leere Wörter kein Problem. Sie erzeugen keinen Eintrag in der flachen Semantik. Für den Generator werden die semantisch leeren Wörter als leere Produktionen aufgefaßt und aus der Grammatik getilgt, wodurch eine neue Grammatik entsteht. Diese enthält dann den von dem getilgten semantisch leeren Wort erzeugten String.

Bei der Elimination der semantisch leeren Wörter können durch das Entstehen unärer rekursiver Regeln Terminierungssprobleme auftreten. Läßt man z.B. in Regel (9)

$$\begin{aligned} &\text{vk} \rightarrow \text{vprt , vk:i} \mid \\ &\text{i:prt= `[vprt:rel}|\text{vk:prt]}, \\ &\langle\text{andere Bedingungen}\rangle. \end{aligned} \tag{9}$$

die Gleichung i:prt=‘[vprt:rel|vk:prt] weg, so wird beim Generieren die Anwendung der durch die Compilation entstandenen rekursiven Regel

vk([E|A],B,(E=ein ; E=mit)) → vk(A,B)

nicht mehr beschränkt und der Generierungsprozeß terminiert nicht. Im Gegensatz dazu terminiert die Generierung bei Verwendung der Regel

vk([E|A],B,F,(E=ein ; E=mit)) → vk(A,B,[E|F])

weil hier von der im Lexikoneintrag (8) vollständig instantiierten Liste vk:prt infolge der beschränkenden Gleichung i:prt=‘[vprt:rel|vk:prt] bei jedem Rekursionsschritt ein Element entfernt wird, sodaß die Regel nur endlich oft anwendbar ist. Falls die Grammatik semantisch leere Wörter enthält, muß der Grammatikschreiber daher selbst für die Kodierung beschränkender Bedingungen sorgen. Der Grammatikcompiler gibt zwar entsprechende Warnungen bei der Entstehung zyklischer unärer Regeln aus, es kann auch vorkommen, daß der Compilierungsprozeß selbst nicht terminiert. Das ist bei Grammatik (10) der Fall.

$$\begin{array}{l} \text{vk} \rightarrow \text{v , vk.} \\ \text{vk} \rightarrow \text{v.} \\ \text{lexicon(Wort,v).} \end{array} \tag{10}$$

10 Constraintaufspaltung bei der Generierung

10.1 Motivation

Während durch den Lexikoncompiler für die Analyse das Lexikon nach der Wortform indiziert wird und beim Parsen die Werte der morphologischen Merkmale daher bekannt sind, indiziert der Lexikoncompiler für die Synthese das Lexikon nach dem Semantikprädikat. Hierbei treten oft zu einem semantischen Prädikat verschiedene Wortformen auf, die in der Form eines disjunktiven Constraints mit der Semantik verbunden sind. Dieser Constraint kann ziemlich groß sein. Vor allem multiplizieren sich bei Anwendung einer Grammatikregel, welche zwei Teilgenerierungen zu einer neuen Teilgenerierung zusammensetzt, die disjunktiven Bedingungen der einzelnen Teile. Die Effizienz der Generierung wird hierdurch beträchtlich verringert. Für den Grammatikschreiber besteht daher die Möglichkeit, durch Angabe gezielter Compileranweisungen Merkmale verzögert auswerten zu lassen. Dadurch sollen disjunktive Constraints erst zu einem möglichst späten Zeitpunkt ausgewertet werden. Konjunktive Bedingungen führen hingegen nicht zur Erhöhung der lokalen Ambiguität.

10.2 Compileranweisungen für verzögerte Constraintauswertung

Geltungsbereich der Compileranweisungen
Derartige Anweisungen sind für Grammatik und Lexikon getrennt möglich. Für das Lexikon können dabei andere Anweisungen wie für die Grammatik gegeben werden, indem diese in die Grammatik- bzw. Lexikondatei geschrieben werden.

Format der Compileranweisung für verzögerte Constraintauswertung
pragma(lazy_gen(⟨Pfad⟩))

wertet ⟨Pfad⟩ verzögert aus.

Hierbei gilt:
Eine Kategorie ist ein Pfad.
Ein Attributpfad ist ein Pfad.
Ist P ein Pfad und A ein Attributpfad, so ist P:A ein Pfad.

Vererbungseigenschaften für verzögerte Auswertung
Ist Pfad1 ein Pfad, so gilt:
Wird Pfad1 verzögert evaluiert, so auch Pfad1:Pfad2 für alle (nach den Merkmalsdefinitionen) zulässigen Pfade Pfad2 .
Wird Pfad1 verzögert evaluiert, so auch Pfad2:Pfad1 für alle zulässigen Pfade Pfad2 .

Wirkungsweise der verzögerten Auswertung
Alle Gleichungen mit verzögert ausgewerteten Attributen sowie Gleichungen mit der Wortvariablen evaluieren zu true. Hierdurch entsteht ein neuer Constraint, der einfacher ist als der ursprüngliche. Der neue Constraint wird nun anstelle des ursprünglichen beim eigentlichen Generierungsprozeß ausgewertet. Ein die Generierung abschließender Test verwendet dann den ursprünglichen Constraint. Wenn der Grammatikschreiber Merkmale, die der Beschränkung von Zuständen dienen, verzögert auswertet, kann es vorkommen, daß Zustände durch die Benutzung des ursprünglichen Constraints ausgeschlossen werden, obwohl sie bei Verwendung des einfacheren Constraints zulässig sind. Dies hat die Meldung
Constraints im Subsubzustand nicht loesbar.
zur Folge und sollte nach Möglichkeit nicht vorkommen, da hierdurch wieder die Anzahl der Zustände bei der Generierung erhöht wird.

Beispiel: Nimmt man in nachstehender Grammatikregel für b:feat den Wert w3 an und versucht, aus b und c a herzuleiten, so wird bei verzögerter Auswertung von feat der Widerspruch erst bei der abschließenden Constraintprüfung festgestellt.

```
a → b , c. |
b:feat=c:feat ,
(c:feat = w1 ; c:feat=w2) .
```

Monotonieeigenschaft der verzögerte Auswertung
Jede Lösung des verzögerten Constraints löst den direkten Constraint.[6]

[6] Dies ist für das Lexikon richtig. In der Grammatik können allerdings Konstituenten bewegt werden. Die Bewegung wird verwaltet, indem eine bewegte Konstituente auf eine Liste gesetzt und in der Landeposition wieder von dieser Liste entfernt wird. Das Entfernen von der Liste kann über eine spezielle Funktion direkt oder verzögert erfolgen. Zweimalige Ausführung des Entfernens ist allerdings nicht möglich. Damit Restriktionen frühzeitig wirksam sind, erfolgt das Entfernen direkt. Der direkte Constraint ist damit kein Teilconstraint des verzögerten Constraints mehr.

10.3 Ein Beispiel

In (2) werden durch Compileranweisungen verzögert ausgewertet:
pragma(lazy_gen(n:morph)).
pragma(lazy_gen(tree)).
Die verzögerte Auswertung von n:morph bewirkt daher die verzögerte Auswertung von n:morph:gen, n:morph:agr:pers, n:morph:agr:num, n:morph:kas:notgen, n:morph:kas:notnom und n:morph:kas:dir. Die verzögerte Auswertung von tree bewirkt die verzögerte Auswertung von n:tree. Dies bewirkt, daß in der Disjunktion die einzelnen Konjunkte zu true, damit die Disjunkte zu true und deshalb auch die ganze Disjunktion zu true evaluiert. Es bleibt daher nur noch ein rein konjunktiver Constraint übrig, der nicht zur Vermehrung lokaler Ambiguitäten beiträgt. Hätte man nur die Anweisung pragma(lazy_gen(n:morph)) gegeben, so wäre die Disjunktion (n:tree = n('Termin') ; n:tree = n('Termins') ; n:tree = n('Termine') ; n:tree = n('Terminen')) erhalten geblieben.

10.4 Evaluation der zusätzlichen semantischen Funktionen

semdelopt- und semdel-Ausdrücke können nicht verzögert ausgewertet werden. Dies würde ihrer Intention als lokale Constraints zuwiderlaufen. Es ist deshalb nicht gestattet, diese Ausdrücke innerhalb von Disjunktionen zu verwenden. Dies führt im übrigen auch zu einer ineffizienteren Verarbeitung, wie folgendes Beispiel zeigt.

Beispiel: Äquivalent sind die beiden Constraints
1)

```
(v:temp=pres, semdelopt([temp(v:var)=pres] ;
 v:temp=past, semdelopt([temp(v:var)=past] )
```

2)

```
semdelopt([temp(v:var)=v:temp] ,
(v:temp=pres ;
 v:temp=past )
```

Bei pragma(lazy_gen(v:temp)), so entsteht aus 1)
1') (semdelopt([temp(v:var)=pres] ; semdelopt([temp(v:var)=past])
aus 2) dagegen
2') true.

In den bisherigen Kodierungen konnte man immer auf spezielle Funktionen in disjunktiven Constraints verzichten.

11 Fehlermeldungen bei fehlerhaften Lexikoneinträgen

An dieser Stelle sind Fehlermeldungen des Lexikoncompilers für den Generator aufgelistet und mit jeweiligen fehlerhaften Lexikoneinträgen versehen, welche diese Fehlermeldung induzieren.

Fehler 1: Cat ist eine Variable
günstiger Fall: es erfolgt eine Fehlermeldung
Beispiel:

```
lexicon(W , Cat) |
       W=bei , Cat=praepkr.
```

Fehler 1: Cat ist eine Variable
ungünstiger Fall/ Nichtterminierung mit talc !!
Beispiel:

```
lexicon(W , Cat) |
       sem(‘sp(Cat:feat1)),
       ( W=wort1 , Cat=kat1
       ;
        W=wort2 , Cat=kat2
       ).
```

Fehler 2
! Spezielle Funktionen dürfen nicht in Disjunktionen auftreten.
Beispiel:

```
lexicon(W,kat) |
       ( W=wort1, semdelopt(‘[x1])
       ;
        W=wort2, semdelopt(‘[x2])
       ).
```

Fehler 3
! Die Semantik darf nur ein einzelner Term, jedoch nicht eine Liste von Termen sein: [wort_grundform(_10834),zusatz(_10834)]
Beispiel:

```
lexicon(W,kat1) |
       sem(‘[wort_grundform(kat1:var),zusatz(kat1:var)]),
       ( W=wort1, kat1:feat1=wert1
       ;
        W=wort2, kat1:feat1=wert2
       ).
```

Fehler 4
! Der Lexikoneintrag enthält mehrere semantische Terme,
unter anderem wort_grundform(_10832) .
Beispiel:

```
lexicon(W,kat1) |
        sem(‘wort_grundform(kat1:var)),
        ( W=wort1, kat1:feat1=wert1
        ;
          W=wort2, kat1:feat1=wert2
        ),
        sem(‘wort_grundform(kat1:feat1)).
```

Fehler 5
! Die Semantik ist eine Variable.
Beispiel:

```
lexicon(W,kat1) |
        sem(‘X),
        ( W=wort1, kat1:feat1=wert1
        ;
          W=wort2, kat1:feat1=wert2
        ).
```

Fehler 6
! f1(_10835)=w1 ist unzulässigerweise keine Liste.
Beispiel:

```
lexicon(W,kat1) |
        sem(‘wort_grundform(kat1:var)),
        semdelopt(‘f1(kat1:var)=w1),
        ( W=wort1, kat1:feat1=wert1
        ;
          W=wort2, kat1:feat1=wert2
        ).
```

Fehler 7
! f2(_10832) ist unzulässigerweise keine Gleichung.
Beispiel:

```
lexicon(W,kat1) |
        sem(‘wort_grundform(kat1:var)),
        semdelopt([f1(kat1:var)=w1,f2(kat1:var)]),
        ( W=wort1, kat1:feat1=wert1
        ;
          W=wort2, kat1:feat1=wert2
        ).
```

Fehler 8
! _10830 ist kein definiter semantischer Term.
Beispiel:

```
lexicon(W,kat1) |
        sem(‘wort_grundform(kat1:var)),
        semdel([f1(kat1:var)=w1,f2(kat1:var),X]),
        ( W=wort1, kat1:feat1=wert1
        ;
```

W=wort2, kat1:feat1=wert2
).

Literatur

[Blo93] Hans Ulrich Block. Compiling Trace & Unification Grammar for Parsing and Generation. In Tomek Strzalkowski, editor, *Reversible Grammar in Natural Language Processing*. Kluwer, Boston, Dordrecht, London, 1993.

[BS92] Hans Ulrich Block and Stefanie Schachtl. Trace & Unification Grammar. In *14th International Conference on Computational Linguistics (COLING-92)*, 1992.

[Hun93] Rudolf Hunze. Botom up generation from flat semantic structures. In *Deklarative und prozedurale Aspekte der Sprachverarbeitung*. Deutsche Gesellschaft für Sprachwissenschaft, 1993.

[Sch94] Stefanie Schachtl. Grammar Development in Trug. 1994.