Studienprojekt Semf
Abschlussbericht

Studiengang
Computerlinguistik und Künstliche Intelligenz

Institut für Semantische Informationsverarbeitung
Universität Osnabrück

Gerd Bongardt, Klaus Dalinghaus, Henrik Dittmann
Michael Huber, Alexander Krumeich, Christoph Peylo
Georg Rehm, Andreas Rother, Wilfried Teiken

Betreuer:
Dr. Lambert Geurts, Dr. Hagen Langer, Prof. Dr. Claus R. Rollinger

Externe Gutachter:
Dr. Peter Gerstl (IBM Deutschland GmbH,
Wissenschaftszentrum Heidelberg),
Prof. Dr. Wolfgang Menzel (Universität Hamburg)

1. April 1997

Dieses Studienprojekt wurde im Rahmen einer Beihilfe des Stifterverbands für die Deutsche Wissenschaft innerhalb des Aktionsprogramms zur Studienzeitverkürzung (Az. 40085/717.2.38) durchgeführt.

Institut für Semantische Informationsverarbeitung
Universität Osnabrück
Sedanstraße 4
49069 Osnabrück

Telefon: +49 541 969 2579
Fax: +49 541 969 2500

Studiengang *Computerlinguistik und Künstliche Intelligenz*
Fachbereich 7, Sprach- und Literaturwissenschaft
Universität Osnabrück
Sedanstraße 4
49069 Osnabrück

Telefon: +49 541 969 2579
Fax: +49 541 969 2500

World Wide Web: `http://www.cl-ki.uni-osnabrueck.de`

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Computer im Sprachunterricht . . . 2
1.2 Studienprojekte . . . 2
1.3 Danksagung . . . 3
1.4 Literatur . . . 4

2 Architektur **5**

3 Die Master-Komponente **11**
3.1 Module Interaction and Dynamic Binding . . . 12
3.2 Function Interface for the Grammar . . . 13
3.2.1 Rule Application Control . . . 13
3.2.2 Functional Expressions . . . 14

4 Der Parser **17**
4.1 Data Structures . . . 18
4.1.1 Numeric Representation of Feature-Value Matrices . . . 18
4.1.2 Typed Features . . . 18
4.1.3 The Implementation of Feature-Value Pairs . . . 19
4.1.4 The `struct`s for Feature-Value Matrices . . . 20
4.1.5 Memory Management . . . 20
4.2 Feature Management Library . . . 21
4.2.1 Functions for Feature Unification . . . 21
4.2.2 Creating Feature-Value Matrices . . . 22
4.2.3 Feature Handling . . . 23
4.2.4 Feature Coding and Decoding . . . 23
4.2.5 Miscellaneous Routines . . . 24
4.3 The Lexicon . . . 25
4.3.1 The Lexicon Interface . . . 25
4.3.2 Internal Structure of the Lexicon . . . 26
4.4 The Parser Kernel . . . 26

4.4.1 Data Structures for the Chart 26
4.4.2 Rule Representation 28

5 Der Grammatikformalismus **29**
5.1 The Parserfile Preprocessor *ppp* 30
5.2 Feature Definition 31
5.2.1 Standard Features 31
5.2.2 Attribute List Features 31
5.2.3 String Features 32
5.2.4 Stringlist-Features 32
5.2.5 Subfeatures 32
5.2.6 Counter Features 32
5.2.7 End of Defintions 32
5.3 The Rule Section 32
5.3.1 `THEME` 33
5.3.2 `FUNC` 33
5.3.3 `GROUP` 34
5.3.4 `LEX-DEF` 35
5.3.5 Additional Notes 36
5.3.6 End of Definitions 36
5.4 The Lexicon Section 36
5.4.1 End of Definitions 37

6 Grammatik **39**
6.1 Syntax 40
6.1.1 Aufbau der phrasalen Kategorien 40
6.1.2 Syntaktische Merkmale 42
6.1.3 Beispiel einer Grammatikregel 43
6.2 Semantik 45
6.2.1 Motivation 45
6.2.2 Funktionalität 48

7 Lexikon **53**
7.1 Format des Lexikons 54
7.2 Das `SYN`-Feature 55
7.3 Das `SEM`-Feature 56

8 Nichtkompositionelle Semantik **59**
8.1 Motivation 60
8.2 Randbedingungen 60
8.3 Umschreibeprozedur 61

8.4 Anaphernresolution 61
8.4.1 Anapherndetektion 62
8.4.2 Stack 62
8.4.3 Beschränkungen 63
8.5 Quantorenbehandlung 64
8.6 Ausblick 64

9 Inferenzmaschine 65
9.1 Inferenzmaschine 66
9.1.1 Schnittstellen 66
9.1.2 Anfragesprache 67
9.1.3 Algorithmus 69
9.2 Sprachbeschreibung 71

10 Weltrepräsentation 73
10.1 Weltrepräsentation 74
10.2 Ontologie 74
10.3 Das Einzelne und die Gruppe 75
10.3.1 Das Einzelne – Entity 75
10.3.2 Die Gruppe – Group 76
10.4 Die räumlichen Relationen 76
10.5 Schnittstellen 79

11 Fehlerbehandlung 81
11.1 Allgemeines 82
11.2 Syntaktische Fehlerbehandlung 84
11.2.1 Einleitung 84
11.2.2 Fehlerklassen in Semf 85
11.2.3 Die Rolle von Grammatik und Fehlergrammatik bei der Fehlerbehandlung 87
11.2.4 Aufbau des `ERROR`-Features 89
11.2.5 Ein paar Beispiele von Regeln 92
11.2.6 Schlußbemerkungen 98
11.3 Semantische Fehlerbehandlung 99
11.3.1 Analyse der Inferenzergebnisse 100
11.3.2 Auswahl des Generalisierungspunktes 101
11.3.3 Durchführung der Generalisierung 101
11.3.4 Bewertung der Ergebnisse 103
11.4 Fehlerbewertung 104
11.4.1 Die implementierte Rückmeldung 104

12 Übung 105

13 Benutzeroberfläche 107
13.1 Einführung . . . 108
13.2 Implementation . . . 108
13.2.1 Aufbau der Benutzeroberfläche . . . 108
13.2.2 Die Übungsseite . . . 109
13.2.3 Die Aufbereitung der Logfiles . . . 109
13.3 Die Oberfläche . . . 111
13.4 Ausblick . . . 113
13.5 Die Visualisierung des Parser-Outputs . . . 114
13.5.1 Syntaxbäume – “tree2gif” . . . 114
13.5.2 Feature-Term-Strukturen – “feat2gif” . . . 115

A Im Verlauf des Projekts entstandene Papiere 119
A.1 Fehler – Transfer – Interferenz . . . 119
A.1.1 Der “Fehler”-Begriff im CALL-Kontext . . . 119
A.1.2 Fehlertypen . . . 120
A.1.3 Interferenzfehler . . . 123
A.1.4 Empfehlung für das Projekt . . . 126
A.1.5 Literatur . . . 127
A.2 Unvollständige Auflistung 127
A.2.1 Aussagen . . . 127
A.2.2 Aktionen . . . 128
A.2.3 Fragen . . . 128
A.3 Vom Semf-System zu behandelnde Fehlertypen . . . 128
A.3.1 Bei der syntaktischen Verarbeitung zu behandelnde Fälle . . . 129
A.3.2 Bei der semantischen Verarbeitung zu behandelnde Fälle . . . 130
A.4 Die Beispielwelt . . . 131
A.5 Zur Koordination von Nominalphrasen . . . 132
A.6 Funktionalität der syntaktischen Fehlerbewertung . . . 133
A.6.1 Funktionen, die über Regelanwendung entscheiden . . . 133
A.6.2 Funktionen, die Werte für das ERROR-Feature berechnen . . . 133
A.6.3 Funktionen nach Beendigung der Satzanalyse . . . 133
A.7 Beispielanfragen an den Inferenzserver . . . 134
A.7.1 Anfragen an die Inferenzmaschine . . . 134
A.7.2 Testsätze . . . 134

B Autorenverzeichnis 137

Kapitel 1

Einleitung

Alexander Krumeich, Georg Rehm

1.1 Computer im Sprachunterricht

Bereits seit einigen Jahren werden Computer im Bereich des Sprachunterrichts eingesetzt. Sowohl im Einzel- als auch im Gruppenunterricht bieten sie sich als Unterstützung oder gar Ersatz zu herkömmlichen Unterrichtsmethoden und -medien an. Besonders in letzter Zeit hat sich auf dem Markt für Lernsoftware viel bewegt, begünstigt nicht zuletzt durch die Möglichkeiten graphischer Benutzeroberflächen und multimedialer Erweiterungen in preiswerten Personalcomputern.

Doch während die Oberflächen der Programme die technischen Möglichkeiten der Computer immer mehr ausnutzen, besteht eine deutliche Diskrepanz zu den Techniken und dem Know-How, mit denen die linguistisch relevanten Bereiche der Programme abgedeckt werden. So bestehen die Übungen in solchen Systemen zum Teil immer noch aus simplen Vokabelabfragen und Lückentexten. In den seltensten Fällen sind hierbei Übungen vorgesehen, in denen ganze Sätze eingegeben und analysiert werden können. Das Vokabular, das der Schüler bei seinen Eingaben verwenden kann, wird meist massiv eingeschränkt. Das liegt hauptsächlich an der Art der verwendeten Fehlerbehandlungsmethoden. Nicht-wissensbasierte System sind darauf angewiesen, daß der Programmautor – der Designer der Übung – eine bestimmte Menge von möglichen Eingaben zu einem bestimmten Eingabefeld *antizipiert* [Krüger, Ronthaler, 1995]. Je geringer die Phantasie des Autors an dieser Stelle ist, desto kleiner wird die Zahl der möglichen Systemreaktionen sein und desto weniger lernerfreundlich wird das Programm sich verhalten.

Bedenkenswert ist weiterhin, daß im Kontext eines Lückentextes oder einer Multiple-Choice-Übung gar nicht die Fülle von möglichen Fehlern auftreten kann, wie sie bei freier Eingabe eines Satzes vorkommen kann. Somit stellt sich die Frage, ob diese Programme überhaupt geeignet sind, den Lerner einer Fremdsprache – entweder unterichtsbegleitend oder autodidaktisch – zu unterstützen.

Obwohl der Computer sich als ein geduldiger Lehrer ausgibt, der sich dem Lerntempo des Schülers individuell anpaßt, ist offensichtlich, daß dieser “Lehrer” meistens nur über sehr wenig Wissen und didaktische Fähigkeiten verfügt. Dies jedoch sind Fähigkeiten, die man von einem Lehrer grundsätzlich erwartet.

Das Projekt Semf stellt den Ansatz dar, Methoden der Computerlinguistik und Künstlichen Intelligenz in den Bereich *Computer Assisted Language Learning* zu übertragen. Mit Hilfe intelligenter Parsing-Strategien, dem Einsatz einer modularisierten Fehlergrammatik und einer semantischen Repräsentation, ist der Semf-Prototyp in der Lage, vielfältige Informationen über vom Schüler gemachte Fehler bereitzustellen.

1.2 Studienprojekte

Die Studienordnung des Studiengangs *Computerlinguistik und Künstliche Intelligenz* an der Universität Osnabrück sieht vor, daß Studierende im Hauptstudium an einem Studienprojekt teilnehmen. Durch die Teilnahme an einem Studienprojekt wird dem Studierenden ein Gefühl für die Teamarbeit in einem industriellen Unternehmen oder einem Forschungsprojekt vermittelt. Damit soll dem Anspruch an das Studium genügt werden, schon während der Ausbildung Praxisnähe zu vermitteln.

In einem Studienprojekt arbeitet eine aus ca. sechs bis acht Studierenden bestehende Gruppe über ein Jahr – zwei Semester inklusive vorlesungsfreier Zeit – an einer vorgegebenen Problemstellung. Die Projektteilnehmer werden dabei von mehreren Dozenten betreut. Des weiteren wird ein Studienprojekt von externen Gutachtern evaluierend begleitet, denen die Fortschritte der Arbeit im Rahmen regelmäßiger Präsentationen dargelegt werden.

Zu Beginn des Projektes wird üblicherweise ein Projektantrag formuliert. Hierbei spielen die Rahmenbedingungen des Projekts eine wichtige Rolle. Zum Teil entstehen in Zusammenarbeit mit Industriepartnern im Rahmen eines Studienprojekts Subsysteme für größere Forschungsprojekte.

Projektname	*Beschreibung*	*Jahr*
PEEP	Präsuppositions-Explikations- und -Evaluations Projekt	1991/92
PROMISE	Projekt mediengestütztes interaktives Sprachenlernen, Englisch	1993/94
SPOCK	Stücklisten-Planungs-Organisation und computergestützte Konfiguration	1993/94
MOPS	Modellierung orthographischer Prozesse	1994/95
SEMF	Semantikbasierte Fehlerbehandlung in CALL	1995/96

Die bislang im Studiengang *Computerlinguistik und Künstliche Intelligenz* veranstalteten Studienprojekte

Am Ende des Projekts steht der Abschlußbericht, der das entstandene System dokumentiert. In diesem Bericht schildert jeder Teilnehmer die von ihm im Laufe der einjährigen Projektdauer erstellten Theorien und implementierten Moduln. In vielen Fällen, z.B. [Bauer, 1995] oder [Krüger, 1995], haben Studenten den von ihnen bearbeiteten Teil des Projektes in ihren Magisterarbeiten weitergeführt.

1.3 Danksagung

Die Teilnehmer des Projekts möchten sich sehr herzlich bei den folgenden Personen für die Unterstützung während der Projektzeit bedanken:

Peter Gerstl, Friedrich Kronenberg, Anja Krüger, Wolfgang Menzel, Dennis Newson und Ingo Schröder.

1.4 Literatur

[Krüger, Ronthaler, 1995] *Intelligente Systemreaktionen in CALL Software*. Universität Osnabrück, Auftragsarbeit für die IBM Deutschland GmbH. 1995.

[Bauer, 1995] *Interferenzfehlerbehandlung in computergestützten Sprachlehrsystemen.* Bauer, Pascal. Magisterarbeit im Studiengang *Computerlinguistik und Künstliche Intelligenz*, Universität Osnabrück. 1995.

[Krüger, 1995] *Interaktives Lernen räumlicher Präpositionen mit dem Sprachlernsystem PROMISE.* Krüger, Anja. Magisterarbeit im Studiengang *Computerlinguistik und Künstliche Intelligenz*, Universität Osnabrück. 1995.

Kapitel 2

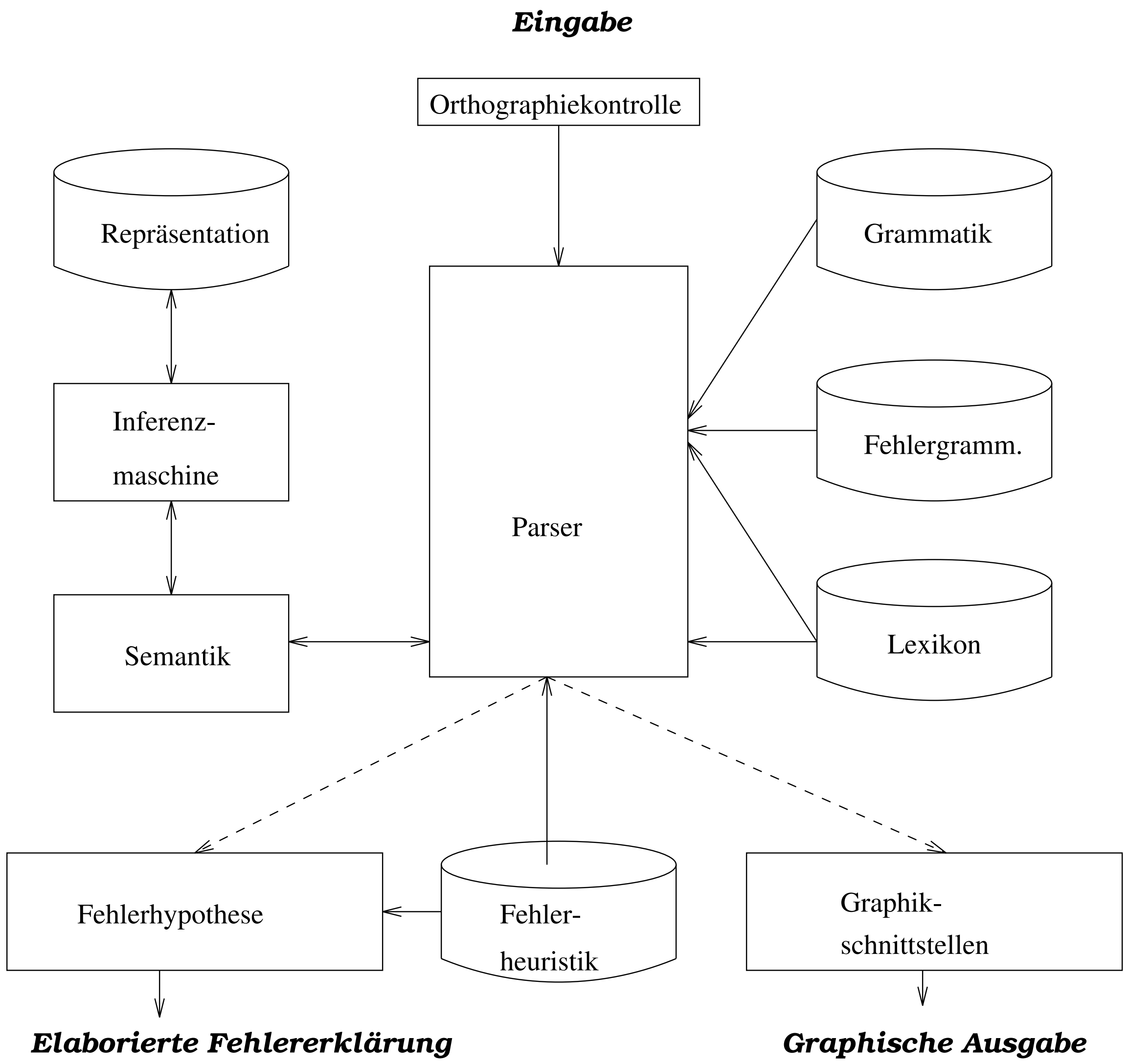
Architektur
Gerd Bongardt

Die Architektur des Semf-Prototypen hat sich im Laufe der Zeit erheblich gewandelt. Die erste Architekturskizze (im Projektantrag) umfasste zwar schon fast alle relevanten Komponenten, mußte aber schon nach kurzer Zeit geändert werden.

Bei der ersten Präsentation wurde schon die Erweiterung um die Masterkomponente vorgestellt. In der folgenden Zeit wurde die Architektur häufiger den tatsächlichen Gegebenheiten angepaßt, was bis zum Ende des Projektes anhielt. Zum Beispiel fielen sowohl die Komponente "Orthographiekontrolle", als auch die graphische Ausgabe aus der Skizze heraus. Die auf der zweiten Präsentation gezeigte Architekturskizze sah deshalb schon so aus.

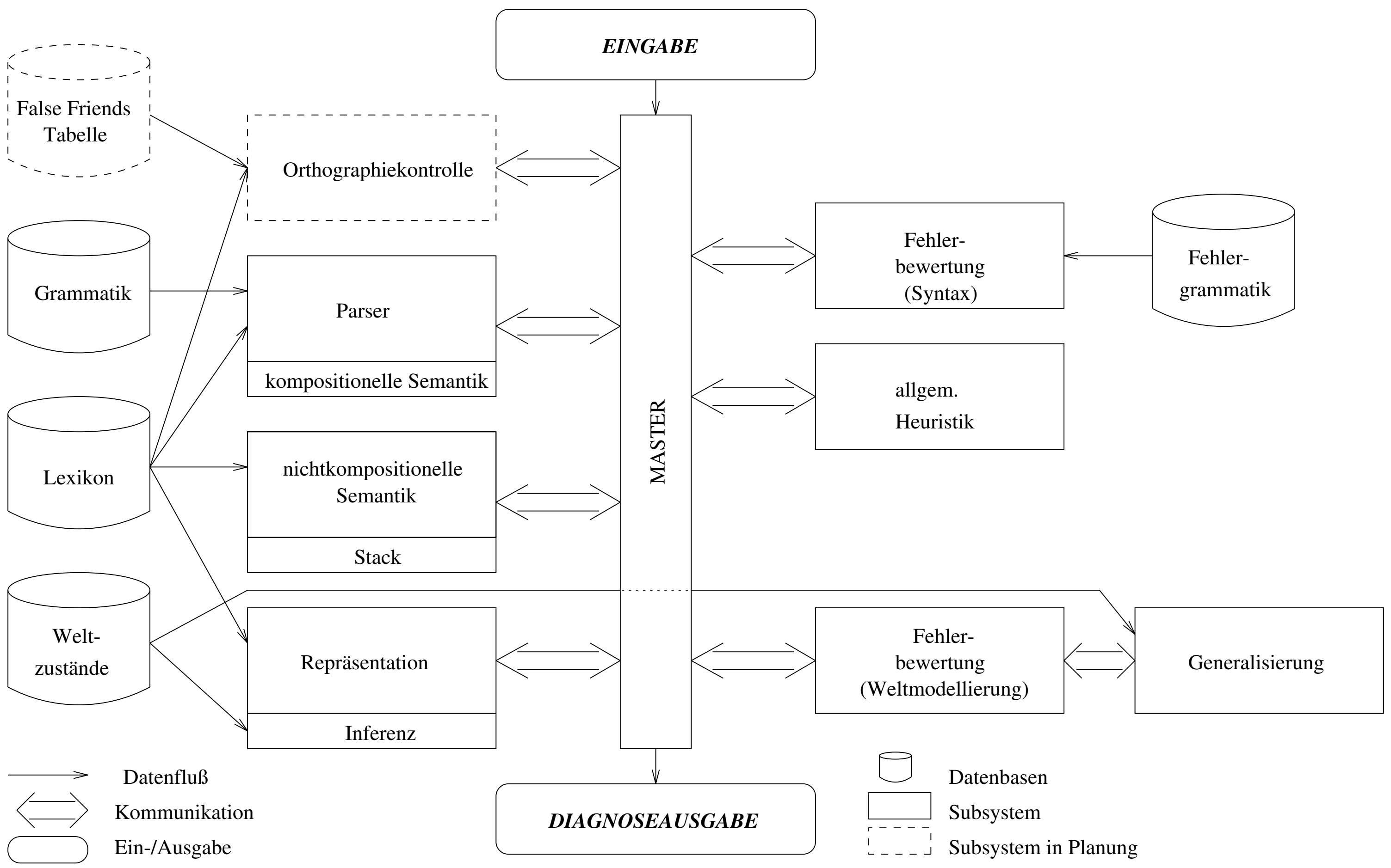
In der folgenden Zeit wurden einige Erweiterungen am Gesamtsystem vorgenommen und zum Schluß, kurz vor der dritten Präsentation, alle Komponenten in eine WWW-Oberfläche eingebunden. Die abschließende Systemarchitektur hatte dann wieder Elemente der graphischen Ausgabe (den WWW-Browser) und mehrere neue Teilkomponenten.

Alle Architekturskizzen zeigen den modularen Aufbau des Semf-Systems, wobei sich zwei grössere Einheiten (Grammatik, Parser, ..., Nichtkopositionelle Sematik, Lexikon und Objekt-Klassen, Inferenzmaschine und Weltrepräsentation, Übungswelten) gerade bei der letzten Systemarchitektur erkennen lassen. Die gesammte Kommunikation läuft über den Master, der sowohl den Input erhält und den Diagnose-Output zum WWW-Browser übermittelt, als auch die Schnittstelle zwischen den einzelnen Komponenten ist. Durch diesen Aufbau der Architektur ist eine Erweiterung, bzw. Reduzierung des Systems ohne erheblichen Aufwand möglich.

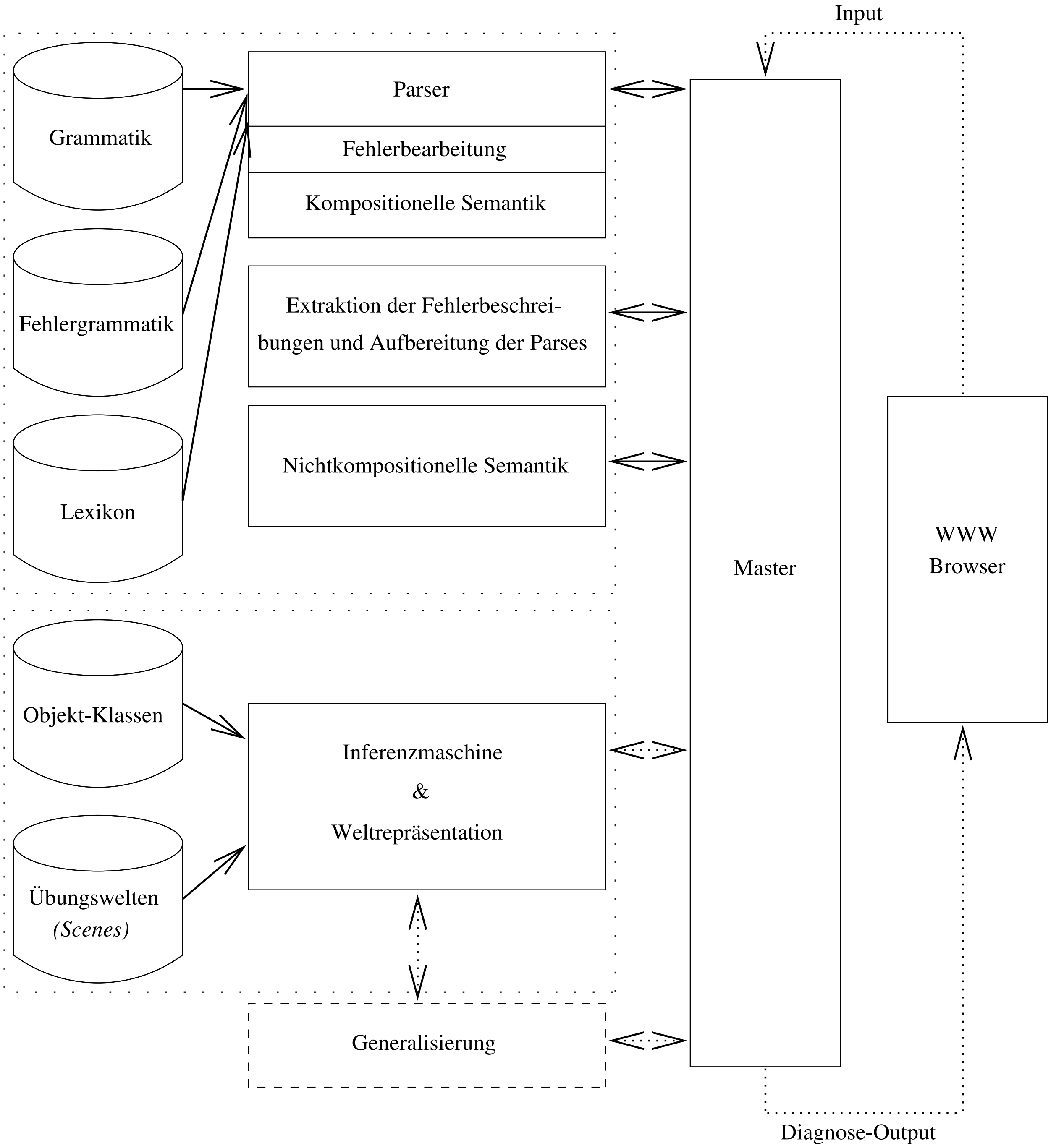


Die Architekturskizze aus dem Projektantrag[1]

[1] In den ersten Skizzen war noch keine Legende vorhanden, was in späteren Versionen dann ergänzt wurde.



Architekturskizze der zweiten Präsentation



Architekturskizze der Abschlußpräsentation

Kapitel 3

Die Master-Komponente

Wilfried Teiken

3.1 Module Interaction and Dynamic Binding

Certain modules used in the parser are connected using a sort of *dynamic linking*. This is done by a component named `master`, its purpose is to call functions in different modules which transform the input. The first called module is the scanner, the second module is the parser. There are two other modules for functional expressions and rule application control. These modules provide functions called by the parser kernel.

Every module has to provide an init function which is called by the master at system startup. Within these init functions the modules normally register themselves using the following function:

dispatch.h, lines 27-27

```
int registerPackage(const char* name);
```

This creates a new package with the name given to the master. If the registration fails, −1 is returned.

For every package, a set of functions may be registered using:

dispatch.h, lines 61-62

```
int registerFunction(int packetnum, const char* name,
                     int data, pblock* (*func)(pblock data));
```

The package number is returned by the `registerPackage` function, it also can be obtained using `getID`. The function is registered for the package named by `id`. The `data` argument takes an integer identifying the type of the parameter the function reveives. These types are conventions taken from `mastertypes.h`, the system does not prevent the programmer from using some identifiers for different data structures. The last argument is the pointer to the function. If the register fails, −1 is returned.

If the module intialization fails, the init function should return zero, nonzero if the initialization is ok.

The registered functions can be called by the name under which they have been registered to the parser. This is especially important for the functional expressions, the grammar only needs a reference to the name of the function, there is no need to store an address or offset for this function. This is necessary to gain independence of parser program changes and grammar changes. The grammar (and the function calls in there) may be changed without recompiling the parser program.

The function arguments are handled using the following struct:

dispatch.h, lines 13-17

```
typedef struct _pblock
{
  int   type;
  void* data;
} pblock;
```

`Type` contains the integer identifier for this field[1], `data` is a pointer to the actual function parameters. Normally, these are structs, the current implementation lists all the used structs in `mastertypes.h`. If the `type` identifyer does not match the stored indentifyer given on function register, the function call will fail.

The function calls are done by a call to:

dispatch.h, lines 79-79

```
pblock* callFunction(int packetnum, int id, pblock data);
```

The parameters are the package number (from `getId`), the function number (from `getFunction`) and a pointer to the argument block.

All the functions return an arguent block struct containing the return type and a pointer to the returned data. The prototype for a function of a module is:

```
pblock* name(pblock data)
```

If a function call fails, `NULL` is returned or the data pointer of the returned `pblock` is `NULL`.

3.2 Function Interface for the Grammar

The parser formalism allows coreferences between features and the result of functions written in *C*. This can be used to access chart information for syntax analysis on the more abstract grammar level. Hereby the programmer can calculate the part of the sentence (the start and the end position) the rule is applied on. As a second approach, functions can be used decide whether a rule should be applied or not, this is used to avoid generation of implausible syntax hypotheses. If a certain hypothesis is not plausible, the rules rules needed to extend the hypothesis to the full length of the analyzed sentence are blocked. This can reduce the number of generated hypotheses and therefore, the computational effort nedded for the parse.

3.2.1 Rule Application Control

The syntax of the grammar rules provides the possibility to call a function before the feature unification for that rule is computed. If `fail` is returned by this function, it appears to the parser kernel as if the unification for that rule has failed.

The rules have to be registred in the `master` module (see 3.1), the parameter struct for this is:

mastertypes.h, lines 42-54

[1] a very simple type check to detect wrong function calls

```
int initFehlerbewertung(void);
#define EVALUATION_REQUEST 20
typedef struct _evalrequest
{
  ITEM*     treeinfo;
  int       count;
  FS**      features;
} evalrequest;
/* FS*-vektor with size */
#define EVALUATION_RESULT  21
typedef int evalresult;
/* plain int */
```

The `treeinfo` pointer is a direct refecence to the `ITEM`-struct corresponding to the rule (see 4.4.1), it contains information about the range of the sentence the rule is applied to. `Count` contains the number of feature-value matrices connected to the rule, every constituent of the rule provides one feature-value matrix, the rule head provides another one.

The pointers to the feature-value matrices are stored in an array pointed to by `features`, the index 0 corresponds to the feature-value matrix of the rule head the indices $1 \ldots n$ correspond to the constituents of the rule. The feature-value matrix with index zero may be modified, although it is recommended to use the functional expressions for this purpose.

The function returns a pointer to an integer, it either zero to inhibit rule application or nonzero to allow rule application.

3.2.2 Functional Expressions

The functional expressions are also *C*-functions called via the `master`-interface. The corresponding parameter-block has the following format:

mastertypes.h, lines 58-69

```
int initFuncexps(void);
#define FUNCEXP_REQUEST 22
typedef struct _funcexprequest
{
  ITEM*     treeinfo;
  int       flags;
  FEATURE** features;
} funcexprequest;
/* zeroterminated FEATURE*-vektor*/
#define FUNCEXP_NEEDVALUE 0x01;
```

```
#define FUNCEXP_OPTIONAL  0x02;
```

These functions are used to gain data which is not accessible by the feature-value matrices or which cannot be calculated using the normal unification rules. All functions return a pointer to a value array for the feature which is terminated by `EOOBJ`[2]. This pointer to the value array may be a cast pointer to a complete feature-value-matrix. If the result of the function cannot be calculated due to e.g. unification failure, the function returns `NULL` or a parameterblock with a `NULL`-pointer instead of the pointer to the value array.

`Treeinfo` is a pointer to the `ITEM` corresponding to the feature-value-matrix the first parameter is taken from. If the first parameter is `$2:SYN`, the pointer is the `ITEM` for the second constituent of the rule. This is the `ITEM` representing the rule from which the constituent was generated. If the constituent is a lexical category[3] an item is temporarily created and filled with the correct information. The function may use the data to access chart information for specific constituents. The function must not modify any of the chart items, all values used by the parser kernel must be created without touching old values. The existing feature-value matrices may of course be used to generate the new (returned) value array, if a complex feature-value matrix is returned.

`Flags` is currently used to store information whether the resulting values are needed for further usage or not. In the latter case it only has to be checked if the function would return a valid feature-value pair. In the other case, the resulting values have to be created and returned.

The last information available to the functions are the arguments given by the grammar. The arguments are stored in an array of `FEATURE` pointers, the list is terminated by `NULL`. There is no fixed argument count, every access to the array has to be validated.

[2]the pointer is returned using a parameter block

[3]there are no entries on the chart for lexical entries, so there is no `ITEM` for them

Kapitel 4

Der Parser
Wilfried Teiken

4.1 Data Structures

This section describes the internal representation of the feature-value matrices used by the parser. It is a short overview of the data structures used by the algorithms to store the feature-value pairs.

4.1.1 Numeric Representation of Feature-Value Matrices

For reasons of efficiency, all features and their corresponding values are stored as integers. All feature and value names are translated to unique numbers, since comparing integers can be done much faster than comparing strings[1].

The translation table is created during compilation of the grammar. Every time the feature declaration part of the grammar is changed, the translation table changes. On the other hand the translation table only changes if the declaration part is changed. Every feature has a unique number, every value has a unique number within a feature, e.g. two values can have the same numeric representation, if they are possible values for different features. The result is that every feature-value pair is represented by exactly one tuple of integers.This representation implies that every possible value for a feature has to be known at compile time of the grammar[2].

The values of string-features are also represented as integers. All strings are treated as atoms in an atom list, the representation corresponds to their list index. Strings can be added to the atom list at runtime, although there is a maximum number of atoms determined at grammar compile time. Usually these dynamically added strins are only used during one parse, so the added strings can be discarded using a library function at the end of the parse (see section 4.2.5). The `RECALL` system uses this feature to generate the strings representing the error position in the sentence.

4.1.2 Typed Features

The current version of the parser uses a typed feature system. Each feature type has special abilities which can be used to achieve various effects during the parse. The currently known feature types are:

`FS_STANDARD` A standard feature. The unification of two possible value sets for features is equivalent to calculation of the intersection of the value sets. If the intersection is empty, the unification fails. If the feature is not present in one of the feature sets, it is treated as the set of all possible values for that feature.

`FS_ATTRLIST` An attribute list feature. The unification of two possible sets of features is equivalent to the join of the two value sets. If the feature is not present in one of the

[1] a name has an average length of about ten bytes, integers have a size of two or four bytes

[2] In the current version there is no possibility to extend the possible range for a feature

feature sets, this set is treated as an empty set for this feature unification. Unification of features of this type cannot fail.

`FS_STRING` A standard string feature. This type is equivalent to the standard type except that the possible value set spans the complete string table.

`FS_STRINGLIST` A string list feature. This is the list feature corresponding to the standard string feature.

`FS_MARKERLIST` A marker list is a special feature for traces of rule application. The unification is equivalent to list concatenation. Unification cannot fail.

`FS_COUNTER` A counter feature. This feature can be used e.g. to count the number of rule applications. The unification of two counter features is equivalent to summing up the counter features. This unification cannot fail.

`FS_SUBFEATURE` A subfeature. This feature can contain a complete feature-value matrix as the value. A unification of two subfeatures is equivalent to unification of every feature present in the unified subfeatures. If one of these recursive unifications fails, the unification of the subfeatures is regarded as failed.

4.1.3 The Implementation of Feature-Value Pairs

A feature-value pair is stored using the following `struct`:

newfeat.h, lines 33-38

```
typedef int *FVALUE;
typedef struct feature
{
  int attr;
  FVALUE val;
} FEATURE;
```

The type of a feature is not stored in the `FEATURE`-struct, it can be determined using the `getType`-function. For a feature with number i, `getType(`i`)` returns the type of the feature.

The `attr`-part of the `FEATURE`-struct is the numeric representation of the feature name, the `val`-part is a pointer to an array of integers in which the values are stored [3]. This array is terminated by the value given as the `EOOBJ` constant. This array is used to represent a disjunction of values as a short form for different possible values[4] or as a conjunctive set

[3] for subfeatures, the pointer is used to store the address of the `FS` (see 4.1.4) containing attribute-value-pairs of the contained attribute-feature-matrix. This counter must be cast to `FS*`

[4] `STANDARD`- and `STRING`-features

of given values[5]. This array is also used for COUNTER-features and MARKERLIST-features. In the first case the first array element contains the counter value[6], the marker list is also a list of items terminated by the EOOBJ marker.

4.1.4 The structs for Feature-Value Matrices

The feature-value matrices are stored as linked lists of FEATURE-structs.:

newfeat.h, lines 42-46

```
typedef struct fs
{
  FEATURE *feat;
  struct fs *next;
} FS;
```

The feat-part of the struct is a pointer to the current feature-value pair (see 4.1.3), the next-part is a pointer to the next possible feature-value pair of the feature-value matrix. The NULL-pointer terminates the linked list. There is a special FS-variable, EMPTYFS, which is used to represent an empty feature-value matrix. This special feature-value matrix may not be modified in any way during the parse, this would cause unpredictable behavior of the parser. Normally, the feat-part of the pointer may not be NULL, but for EMPTYFS, this is not true. Therefore check feat every time before accessing it.

4.1.5 Memory Management

The dependencies between the different feature-value pairs and feature-value matrices created during a parse are rather complex. These dependencies are not traced in any way, so in general there is no possibility to figure out whether a certain value array, feature-value pair or feature-value matrix is currently used or not. The linked lists may be pointed to by other feature-value matrices at any start point, the value array may be used partly.

Due to these facts, modifications of existing features or value arrays are extremely dangerous, only the new generated items should be modified (or in this case, initialized). It may be wise to use special functions to allocate memory, as it is done within the parser. The used function is mymalloc, or one of its higher level interfaces, safemalloc and blockedalloc. All memory allocations are tracked in a database, so that these memory blocks can be freed as soon as the complete set of feature-value matrices is not further required. All memory allocated for one parse can be freed with one call of free_at, this function uses the tracked memory allocations. During the parse, the memory is normally

[5] ATTRLIST and STRINGLIST

[6] there has to be an EOOBJ marker for counters, too

not freed[7]. In further versions, there may be a garbage collector. In fact, this leads to additional overhead in memory management. It is left to real time applications to show whether the reduction of memory requirements at parse time is worth the extra cpu requirements caused by a garbage collector.

4.2 Feature Management Library

4.2.1 Functions for Feature Unification

To calculate the results of a unification, the *RECALL* parser offers a library of functions. These are described in the following.

newfeat.h, lines 58-67

```
FVALUE    valueCounter(FVALUE v1, FVALUE v2, int update);
FVALUE    appendValue(FVALUE v1, FVALUE v2, int update);
FVALUE    concatValue(FVALUE v1, FVALUE v2, int update);
FVALUE    unifyValue(FVALUE v1, FVALUE v2, int update);
FEATURE*  unifyFeature(FEATURE *f1, FEATURE *f2, int update);
FS*       unifyFS(FS *fs1, FS *fs2, int update);

FS**      unifyFSvectors(FS** FSarray1, FS** FSarray2, int count);
FS*       specializeFS(FS* source, FS* with,
                       int spath, int path, int update);
```

The last argument is always an integer indicating whether the result of the unification is needed (non-zero value) or not. In the latter case, it is only checked whether the unification would fail (a `NULL`-pointer is returned) or not (a non-`NULL` pointer is returned).

`ValueCounter` and `appendValue` are low-level functions to create value-arrays treated as sets. Given the pointers to two sets, they calculate the intersection respectively the join of the two sets. The array containing the set is allocated via `safemalloc`, therefore it should be freed by `safefree` or preferably by `free_at` If any of the input sets is equal to the result, a pointer to that array is returned instead of creating a new array. The third function, `appendValue`, is the low-level-function for `MARKERLIST`s. It simply creates a new array containing the concatenation of the two lists. Using these functions, the feature type is not checked, so the application of the wrong low-level function may cause unintended behavior of the parser. In context of non-parser-kernel operations it is recommended to use the higher level of the library.

The `unifyFeature`-function chooses the correct low-level function to calculate the correct unification for two given feature-value pairs. If the two features have different attributes, the result may be unintended. There is no test in this function, because there may

[7] exceptions are handled via `safefree`, which updates the corresponding entries in the memory allocation database

be circumstances, in which the unification of two different (but related) features may be necessary. There is a check for the type of the two features, you may not unify two different feature types, this attempt will always fail. This function may call the `unifyFS`-function recursively if the given parameters are `SUBFEATURE`s.

`UnifyFS` operates on an even higher level. In this case the arguments are complete feature-value matrices, the result is a new feature-value matrix representing the most specific feature-value matrix subsumed by the given feature-value matrices. This feature-value matrix may share parts of memory with both given feature-value matrices. The `unifyFSvectors`-function is a variant of the `unifyFS`-function, it is used to unify several pairs of feature-value matrices with one function call. The corresponding feature-value matrices in the two lists are unified, and this list of feature-value matrices is returned. The result is always calculated, there is no *update*-parameter for this function. The memory used to store the pointers to the feature-value matrices is owned by the function, so do not attempt to free this memory in any way.

The last function in this section, `specializeFS`, is a direct representation for the coreference of feature-value pairs in given feature-value matrices. This function returns the most specific feature-value matrix which subsumes the first feature-value matrix and a part of the second feature-value matrix. This part is referenced by the second path, respecting the first given path as a *destination* for the coreference. See section 5.3.4 for details. This function is a straightforward implementation of the described mechanism of coreferencing feature-value pairs.

4.2.2 Creating Feature-Value Matrices

newfeat.h, lines 75-77

```
FS* makeFS(int lexnum, int staticmemory);
FS* lex2FS(int offset, int staticmemory);
FS* copyFS(FS *source, int staticmemory);
```

These functions are used to create new feature-value matrices. They are normally used every time a chart item for a lexical category is created. The function `makeFS` creates a new feature-value matrix for a given lexical item. When called, `MakeFS` receives the number in the lexicon as its first argument.

Lex2FS has similar purposes, the only difference is that its argument is a direct offset to a list of value-arrays created at compile-time of the parser. These offsets are stored in internal tables. Usually this functions is only called by the parser kernel. It provides the low-level services for `makeFS`.

The last function in this library part is `copyFS`, it creates a copy of a feature-value matrix. It is important that only the control information for the linked list, the `FS`-structs, is copied. The value parts are shared between the original and the copied list.

The second argument is always a reference whether the feature-value matrix is nested for this parse only (value 0) or for the complete parser runtime, which may span multiple parses (non-zero value). The latter case is used e.g. for feature-value matrixes for postfix expressions in the lexicon. This is only done for improvement of efficiency.

4.2.3 Feature Handling

newfeat.h, lines 85-87

```
FEATURE* findFeatureByNum(int attr, FS* tosearch);
FEATURE* findFeatureByPathNum(const int* patharray, FS* tosearch);
FEATURE* findFeatureByPath(const char* pathstring, FS* tosearch);
```

These three functions can be used to locate feature-value pairs in feature-value matrices. `FindFeatureByNum` locates a feature-value pair in a feature-value matrix, it is located by its attribute number. If no feature-value pair is included in this feature-value matrix, a `NULL`-pointer is returned. The feature-value matrix is not searched in full depth, only the top level features are used.

`FindFeatureByPathNum`locates a feature by its path, given as an integer array. It is equivalent to an iterative call to `findFeatureByNum`, descending in the encapsulated feature structure. The array is terminated by `EOOBJ`. If one of the features is not a `SUBFEATURE`, `NULL` is returned. `NULL` is also returned if one of the features mentioned in the path cannot be found.

`FindFeatureByPath`has the same purpose. In this case the argument is given as a string, not in its numerical form. Names in the path are seperated by `->`, e.g.: `SEM->OBJECTINFO->QUANTIFIER`.

4.2.4 Feature Coding and Decoding

newfeat.h, lines 95-105

```
int getFeatureNum(const char* featureName);
int getAttrNum(int attr, const char* attrName);
int getStringNum(const char* string);

char *decodeFeature(int attr, int val);
const char* attrName(int attr);
const char* attrValue(int attr, int value);
int getType(int attr);

void printFeature(int file, int attr, FVALUE val);
void printFS(int file, FS *);
```

The first three functions can be used to determine the numeric representation of features or values if the names are known. `GetFeatureNum` gets the numeric representation for a feature of a given name. `GetAttrNum` returns the numeric representation for a value of a known attribute. `GetStringNum` returns the number of a string in the string list. Every `STRING`-feature can have any string as a value, so there is no need to handle different numbers for different features as it is necessary for `STANDARD`-features.

The next block of functions is used to get the string representation of features, if the numeric representation is known. The memory for the strings is owned by the functions, so it may not be freed in any way by the calling functions. `decodeFeature` returns a string of the form "FEATURE=VALUE", it can be used to print a feature-value pair on the screen. `attrName` returns the name of a feature, `attrValue` the name of a value of a certain feature. This also works for `STRING` and `COUNTER` features, but not for `SUBFEATURE` features.

The function `getType` returns the type of a given feature. The following possible values are:

newfeat.h, lines 16-22

```
#define FS_STANDARD 1
#define FS_ATTRLIST 2
#define FS_SUBFEATURE 3
#define FS_COUNTER 4
#define FS_STRING 5
#define FS_STRINGLIST 6
#define FS_MARKERLIST 7
```

The last two functions print a dump of a feature-value pair of a feature-value matrix on a given file handle. These file handles are not related to the operating system, the are only related to the parser, they are handled internally.

4.2.5 Miscellaneous Routines

newfeat.h, lines 109-109

```
int addString(const char* string);
```

The `addString` function adds a string to the string list. If there is no more space in the string table, -1 is returned. If the string is already in the string table, no second copy is stored. The function returns the number of the string in the string table.

The string is copied when it is stored, therefore the original string may be discarded, if it is stored in dynamically allocated memory. Also, a string in a static memory location may be given to the function, the memory location can be reused by other functions.

4.3 The Lexicon

The lexicon module is the interface to lexical entries for the parser. The current implementation includes a lexicon with some morphologic capabilities. The current version of the lexicon is written manually, but the interface can be used to access a standard lexicon backend. The lexicon routines are based on a word by word lookup of lexical entries. The entries are separated by whitespace.

The Lexicon module has to create the feature-value matrices for the parser, the current implementation uses the functions `makeFS` and `lex2FS`, see 4.2.2.

4.3.1 The Lexicon Interface

The interface between the parser kernel and the lexicon is defined by a small set of functions: lexicon.h, lines 20-29

```
typedef struct sempointerlist{
  YYSEMTYPE *fs;
  int cat;
  struct sempointerlist *rest;
}SEMPOINTERLIST;

SEMPOINTERLIST *get_le(const char *word, int lowercase);

int readLexicon(FILE* infile);
int initLexicon(const int* tails, const int* tailref, int lexCount);
```

The function `get_le` is called to get the lexical information (category and the initial feature-value matrix) for the given word. A linked list is returned, every item in the list representing one possible interpretation for the word. The current implementation of the parser has no support for lexical entries spanning two words[8]. The linked list consists of `SEMPOINTERLIST` entries, each entry includes the feature-value matrix in `fs`, the category for this word in `cat` and a pointer to the next entry in `rest`. A `NULL`-pointer in `rest` terminates the list. If the word is unknown, `get_le` returns `NULL`.

The other two functions are used to initialize the lexicon module. `ReadLexicon` is called when the lexicon section is encountered while reading the parser initfile. The parameter is a pointer to the `FILE`-structure. A blank line read terminates the lexicon section. If the lexicon interface is used to connect to an external lexicon, this function call should return without any action. A non-zero value must be returned in case of correct initialization. `InitLexicon` is called after the complete parser data structures are initialized[9]. This function call can be used to initialize an external lexicon interface. The parameters are pointers to tables

[8] the only possible solution is to transform the two-word-input to one word using the integrated scanner
[9] the chart is not yet initialized

indicating possible suffix inhibit rules for suffix entries and from suffixes inhibit rules to suffixes. For details, see 5.4. These arguments may be ignored if the internal lexicon is not used. This function must return a non-zero value, if the initialization of the parser is correct.

4.3.2 Internal Structure of the Lexicon

The lexical entries are stored in three different *btrees*, one for normal lexical entries, one for root entries and one for suffixes. There are pointers from the suffixes to the corresponding suffix inhibit rules and from the inhibit rules to the corresponding regular suffixes (which may in this case *not* be applied). These pointer tables are created at compile time of the grammar, they are stored as `tails` and `tailref`. These lists are the arguments for `initLexicon`. Every time a word is looked up, it is first searched in the btree for normal lexicon entries, then as string start in the root-btree. If any matching entry in the root-btree is found, the corresponding entries in the suffix-btree are searched. Every matching suffix is checked, if the complete word is mentioned in an inhibit rule for this suffix. If this is the case, the corresponding error-feature-value matrix is unified with the feature-value matrix of the word entry and the suffix entry. In the other case, the feature-value matrix of the word entry and the suffix entry are unified.

The lexical entries used in inhibit rules are found in the *normal* btree. If this entry is used, the corresponding feature-value matrix is calculated using the feature-value matrix of the corresponding root entry and the corresponding suffix entry. See 5.4 for details of this mechanism.

4.4 The Parser Kernel

The complete feature handling is done with the routines described in 4.1, so this chapter will concentrate on the chart structure.

4.4.1 Data Structures for the Chart

The central data structure for the char is a chart item:

epcore.h, lines 41-48

```
typedef struct item{
  int from;
  int to;
  int derived;
  RULE rule;
  int flag;
  struct structp *struktur;
```

```
}ITEM;
```

This structure uses a second structure to connect different chart items together:

epcore.h, lines 52-56

```
typedef struct structp{
  int fromnode;
  int fromitem;
  struct tree *fromp;
}STRUCTP;
```

The chart is organized as an array of arrays of items. The first dimension represents the nodes of the item, the second dimension the actual item number for this node, e.g. `ch_node[3][10]` is the tenth chart item for the third node. The node with the number zero is the node on the right side of the first word, therefore it may only be filled with items with their dot at the very beginning of the constituent list.

The chart includes the position of the item (`from` and `to`, the start point and the end point of the item), a pointer to the item representing the last found constituent of this item (as it must be in the same node, the index is sufficient to identify the item completely). `RULE` is a pointer to the actual position in the grammar rule for this item[10], see 4.4.2 for details.

The flag is used to identify whether the rule was succesfully applied (value `RULE_ACTIVATED`) or not (value `RULE_PREPARED`). This is used to decide whether the unifications necessary for this rule were succesful. The pointer to the `structp` is used to connect the items for a rule.

The `fromnode` and `fromitem` identify the item for the right neighbor constituent of this rule. If not present, the pointer to the `STRUCTP` is `NULL`. The `fromp` pointer points to the tree information for this node, this includes the subtrees for this node and the feature information for this node as well as the category for this node. This information is used to print out the syntax trees and the feature information at the end of the parse.

The tree information is stored in the following struct:

epcore.h, lines 60-69

```
typedef struct tree{
  int kategorie;
  YYSEMTYPE *value;
  struct treelist *dtrs;
}TREE;

typedef struct treelist{
  TREE *first;
  struct treelist *next;
}TREELIST;
```

[10] in other publications often referenced as the *dot*

The first data structure contains the category and the feature-value matrix of the node item. The next data field contains a pointer to a linked list with the daughter items for this node.

The complete parses are stored in an array of following structs:

epcore.h, lines 73-76

```
typedef struct complete_parse{
  int to;
  TREE *tree;
}COMPLETE_PARSE;
```

The `to`-part contains the last item of the current parse (which is not necessarily the last item of the input, this has to be checked), the `tree`-part contains a pointer to the `TREE` information of the initial symbol, normally `S`.

4.4.2 Rule Representation

The rules are stored as an integer array. Every category is translated to an integer, the rule is an array including the head at index zero and the categories at the following indices. The list is terminated by `STOP`. The chart items contain pointers in this integer list, the pointer points to the offset in the array with the currently unscanned category. The test whether the item is found or not is done by comparing the scanned category with the dereferenced `rule` pointer. The `START` symbol has the number zero, so the first head of the first rule in the grammar file has to be the top symbol of your lexical system, normally `S`.

Finally, there are a few definitions which are useful to understand the parser kernel:

epcore.h, lines 34-37

```
#define START S
#define YYSEMTYPE FS
typedef int *RULE;
typedef RULE *GRAMMAR;
```

Kapitel 5

Der Grammatikformalismus
Wilfried Teiken

The parser package includes a compiler for a parser, *gramr*, which is used to translate the ASCII-file containing the grammar and lexicon description into a precompiled format. This file only contains integer values describing the actions corresponding to the described grammar and some translation tables. The current version of the *RECALL*-system uses multiple files for the parser input. The main file is preprocessed by *cpp*, the *C* preprocessor, and includes the other files via #include-statements. As the parser source is preprocessed via *cpp*, all *C* macro features are available.

5.1 The Parserfile Preprocessor *ppp*

The parserfile is preprocessed by the internal preprocessor *ppp*. This preprocessor creates a hierarchy of rules out of one single rule using some metatags. These tags mark certain coreferences in the rule as *optional*, if they are omitted, some information about this omission is aded to the feature-value matrix. Ppp links the rules with special tags, this guarantees that only the most specific rule of a given set will be applied. It may be the case that there is more than one rule of an equal level of specialization, in this case all three rules are applied as concurrent alternatives. This mechanism is mainly used for automatic generation of error rules for e.g. agreement errors.

The syntax for the tags is: @ <tag> { <variant code> }
$$+ <tag> { <code included if tag is present in rule> }
$$- <tag> { <code included if tag is not present in rule> }

Every block with an @ defines one additional variant of the rule, one with this block included in the rule body, one without it. If multiple tags are included in one rule, every possible combination of tags is created, e.g. two tags create four rules: one with both, one with the first, one with the second and one without any tags. As a second step the rule is scanned for connected blocks in the rule: if a tag *foo* is included in the rule, every block marked with $$- foo is deleted from the rule. The blocks tagged with $$+ foo are included in the rule, but the tag itself is removed. If the tag is omitted in a rule, the connected blocks rules flip.

The metatag * is used to figure whether any tag is omitted[1] or not[2]. This is used to include error rule specific parts into the rule automatically using the $$- *-tag. The *-tag is predefined, therefore it may not be used as a tag name.

Due to the fact that not all combinations of tags are useful, there is a possibility to control the tag omitting mechanism. In front of any rule there may be added some lines of the following form:

<name_1> | <name_2> { | <name_n>}

These lines create the constraint, that at least on of the tags name_1, ..., name_n must be included in every generated rule. Every line creates on constraint, the constraints must

[1] in ths case, all blocks tagged with $$- * are included in the rule
[2] if all tags are included, the blocks tagged with $$+ * are included into the rule

hold on any created rule. This prevents the mechanism from generating useless rules. There must not be any empty line between the constraints.

The last feature of the preprocessor is the *variable* `$c`, every occurance of `$c` in the rule is replaced by the number of omitted tags. The *RECALL*-grammar uses this for the generation of the `ERROR->COUNT` feature.

5.2 Feature Definition

The input file for the parser generator must start with the feature definitions. Every feature used in the grammar or lexicon has to be defined in this section. The definitions include the type and the range of the feature:

```
(#<type> <name> <value_1> ... <value_n>)
```

`Type` is the feature type, `name` is the name for the feature[3] and the `value`-entries are the possible values for this feature. Not every feature definition includes these values, as seen below. The semantics of the different types is given in section 4.1.2.

5.2.1 Standard Features

```
(#attrdef <name> <value_1> ... <value_n>)
```

These features are used to represent *normal* features. The possible values are the values given in the definition, a disjunctive set of values may be given to represent a possible range for that feature as a value. As an example, the `NUM` feature may have the values `SING` and `PLUR`, for some entries with identical forms for singular and plural entries in the lexicon, the disjunction set of these two values may be used to represent that both values are possible[4] (see 4.1.2, `FS_STANDARD`).

5.2.2 Attribute List Features

```
(#attrlistdef <name> <value_1> ... <value_n>)
```

This feature is used to represent a set of features, in which every value is valid. The set does not represent multiple possibilities, it is used to accumulate different attributes of an object. For example, if the possible values for a feature `ATTRIBUTES` are `SMALL`, `BIG`, `RED` and `BLUE`, the set `RED, BIG` describes a big, red object[5](see 4.1.2, `FS_ATTRLIST`).

[3]these names have to be unique

[4]However, not both values are valid at the same time, the two possibilities exclude each other

[5]The parser does not check for semantically exclusive feature sets, a `BIG, SMALL` object is valid for the parser

5.2.3 String Features

`(#strdef <name>)`

This feature is used to store strings. These strings can include the charset *[*`"%\'\~A-Za-z0-9_`*]*. Every string used in the grammar and lexicon part of the input file is a possible value for any string feature. There is no need to define the strings a priori in the feature definition(see 4.1.2, `FS_STRING`).

5.2.4 Stringlist-Features

`(#strlistdef <name>)`

These feature type is the list version of the string feature (see 4.1.2, `FS_STRINGLIST`).

5.2.5 Subfeatures

`(#subfeature <name>)`

Subfeatures can store a complete feature-value-matrix as its value. Every valid feature-value-matrix is a valid value for this feature, so there is no need for value definitions(see 4.1.2, `FS_SUBFEATURE`).

5.2.6 Counter Features

`(#counter <name>)`

Counter features can be used to store integer values(see 4.1.2, `FS_COUNTER`).

5.2.7 End of Defintions

The feature definition part of the input file is terminated by the following statement:

`%eoattrdefsq`

This statement has to occur on a single line.

5.3 The Rule Section

The rule section of the input file begins with

`%rules`

The rules follow this statement, the basic syntax for grammar rule definition is:

```
<HEAD> : <NT_1> {<NT_n>} [&<THEME1>] {<THEMEn} [*<FUNC>]
                  [#<GROUP1>] {<GROUPn>}
       [
```

```
        $0:[     <LEX-DEF_1>;
                 <LEX-DEF_m1>;
         ]
        $1:[     <LEX-DEF_2>;
                 <LEX-DEF_m2>;
         ]
        ...
        $n:[     <LEX-DEF_n>;
                 <LEX-DEF_mn>;
         ]
]
```

A rule definition consists of a PSG part naming the lexical categories and a part including the coreferences for the feature structure. The coreference part is based on the *PATR* formalism.

The coreference part is divided in several sections, one for every constituent mentioned in the PSG-part of the rule. Each of these parts may be omitted, if there is no need for coreferences for certain constituents. The coreferences following the `$0:` are the coreferences done for the rule head, the other coreferences are applied to the rule constituents. Only the coreferences mentioned in the head part are stored and used for the further parsing. The coreferences given in the other parts (`$1:` to `$`n`:`) are merely checked for validity, but the corresponding feature-value matrices are not modified.

5.3.1 THEME

Upon the parsing of a sentence the type of the sentence (e.g. question or statement) may be known. Therefore, a part of the grammar may be considered useless or even disturbing for that parse. It is thus desirable to exclude certain rules to increase performance and to avoid misleading hypotheses. This is accounted for with the `THEME`. Every rule may be tagged with a `THEME`, the rules tagged with the same `THEME` form a unit[6]. Each `THEME` may be disabled for a certain parse, if no `THEME` set is given for a parse, the set defaults to the `THEMES` necessary to parse simple statements (no questions or imperatives). A rule is disabled, if every `THEME` it belongs to is diabled. If a rule is needed for different types of sentences, tag it with all needed `THEMES` for these sentenes.

5.3.2 FUNC

A `FUNC` is the name of a function used to control rule application (see 3.2.1). The functions are written in *C*. These functions can be used to cut off implausible syntax hypotheses. If the plausibility for a certain hypothesis does not reach a given threshold, the function

[6] a rule may have multiple `THEME` tags, this means the rule belongs to multiple groups

can cause the unification to fail[7]. The function has access to the features and to the chart items to decide wether a rule should be applied.

5.3.3 GROUP

The `GROUP`s link rules together. A rule which is applied at a given point during the parse may disable its linked groups for the rest of the parse to save processing time and to avoid superfluous syntax hypotheses.

The standard application for this are the agreement error rules used in the parser. These rules form a subsumption hierarchy. For the standard rule there exists a set of error rules with one false agreement. A second set of rules exists, in which every rule has two agreement errors. This may be continued to the maximum number of agreement errors which can occur with a certain rule. If the correct rule is applied, every rule with agreement errors may be applied, too, because the coreferences for the error rule is a subset of the coreferences belongig to the standard rule. The same scheme applies to an error rule and its less specific error rules. The preprocessor (see 5.1) generates tags for the rules, which links the more and the less specific rules.

If the first rule of a group is applied for a certain part of the sentence (e.g. from the beginning to the end of the second word), the application of every linked rule *on the same part of the sentence* is blocked. This represents the subsumption hierarchy of the rules, in which every rule is linked to all the rules which are less specific. An important point is that only the first rule of a set, the master rule, may disable the other rules of the set, called slave rules.

A rule may be a slave rule to multiple masters, this is used to represent the different levels of the subsumption hierarchy, a rule is a slave rule to all more specific rules in the subsumption hierarchy[8].

Extended Master Rules In certain cases, there is no strict subsumption hierarchy and the mechanism used for the agreement errors cannot be applied correctly. The main problem which occured, when the grammar was written, is the need to have more than one rule which can disable the rest of the rules in the set. The solution implemented in the parser is the *extended masterrule*, the syntactical marker for this rule is a `+` (plus sign) at the beginnig of the group name. Every (extended) master rule blocks the slave rules of the set, but the extended master rules do not block each other. If the *real* master rule is applied, all extended master rules are disabled.

[7] Actually, the function causes the unification to be blocked to save processing time

[8] Every rule in the subsumption hierarchy gets a unique label as a group tag, every less specific rule gets the same tag to link it as a slave

Extended Slave Rules Another problem occured using the master rules when the grammar got more complex. If a large set of hypotheses is calculated, the conditions for the application of the rules, which are order dependant, are too weak. Superfluous parses are generated in these cases, on the other hand parses get lost. As a result, the mechanism of extended slave rules was implemented. The extended slave rules are marked by a `!` (exclamation mark) at the begining of the tag name. If a rule application fails due to unification, the extended slave rules are tried (the extended slave rules are not put into the chart before that). This guarantees that the rules are not applied if the master rule succeeds. If the applications of one of the slave rules fails, the extended slave rules for that rule are tried. This mechanism is continued iteratively. As the application test is done level by level, only the most specific rules are applied to parse the sentence (the subsumption hierarchy for the rules is used to generate links similar to the mechanism used fo the normal slave rules). As a second effect the chart is smaller, because the extended slave rules are only put into the chart if the master rule fails.

In the current version of the parser most links are done using the extended master rules, these links are generated by the preprocessor.

5.3.4 LEX-DEF

The `LEX-DEF` part of the rule contains the coreferences needed for a rule application. The valid coreferences are:

```
<name> = <value>;
```

A simple (static) assignment of a value for a feature.

```
<name> = {<value_1>...<value_n>};
```

A set of values assigned to a feature. Depending on the feature type this set is disjunctive or conjunctive. In the first case the feature-value matrix is a short form for two (or more) possibilities excluding each other or it is an accumulative list pf properties.

```
<name> = $n:<name>;
```

A coreference of feature `name` with the feature `name` in the feature-value matrix of constituent n (or the rule head, if $n = 0$).

```
<name> = $n:<name_1> -> ... -> <name_n>;
```

A coreference of the feature `name` with the feature `name_n` in subfeature `name_(n-1)` ... in the subfeature `name_1` of the feature-value matrix belonging to the constituent n. Every feature of the set `name_1`, ..., `name_(n-1)` has to be a subfeature. This sequence of subfeatures connected by the `->` operator is called a path.

```
<name> = <funcname>(<path_1>, ... ,<path_n>);
```

The C-function `funcname` is called. It receives the feature-value-pairs pointed to by the paths `path_1`, ..., `path_n` as its arguments. The result returned by the function is used as the new feature value (see 3.2.2).

5.3.5 Additional Notes

A feature my be mentioned multiple times in a coreference list, the last assignment defines the actual value for the feature. However, the value given in the first assignment can be used in the second assignment[9]. This is done to assign initial values to counter features:

```
[
  COUNTER = 3;
  COUNTER = $1:COUNTER;
]
```

This assigns the initial value 3 to `COUNTER`, the next assignment increases `COUNTER` by the value of feature `$1:COUNTER`. The static assignments are always executed before any other assignment is done.

The coreferences may be nested, this is used to set the value of subfeature elements. A sample coreference is:

```
   [ attribute_a = [
                    attribute_b = value;
                   ]
   ]
```

5.3.6 End of Definitions

The rule section is terminated by

`%eorules`

5.4 The Lexicon Section

The lexicon section is started by

`%lexicon`

An entry in the lexicon has the following format:

```
   <word> <category> [
                       <LEX-DEF_1>;
                       <LEX-DEF_m1>;
                     ]
```

[9]This means the result of the coreference set is order dependant

The `LEX-DEF`s are similar to the ones in the rule section, except for the fact that coreferences and functional expressions are not allowed.

The lexicon has a simple morphologic capability. Lexicon entries which use this feature must comply with the following notation:

If word ends with ~ (tilde sign), it is treated as a root entry for the lexicon, if it begins with ~ (tilde sign), it is treated as a suffix (see 4.3.2). The suffix entries have a special syntax:

```
~<suffix>:<name> <category> [ ... ]
```

The first part is the suffix, the `name` following the colon is a special tag used to identify the suffixes in the suffix inhibit rules. The feature-value matrix given for the suffix is unified with the feature-value matrix of the root entry belonging to the suffix.

The format of an suffix inhibit rule is:

```
<stem>/<name>~<suffix> <category> [ ... ]
```

`Stem` is the root of the word, e.g. *sandwich*. The name is the tag identifying the erroneous suffix, e.g. `N_PLUR_SUF`, which means the suffix *~s*. This suffix, which is correct for regular nouns, is inhibited for the stem `sandwich`, if the word **sandwichs* is used it is recognized, but the feature-value matrix belonging to the inhibit rule is unified with the feature-value matrices gained from the word stem and the *incorrect* suffix.

`Suffix` is the correct suffix which has to be used instead of the inhibited rule, in this case this would be *es*. This mechanism is used e.g. to handle mostly regular verbs which are irregular in some cases.

5.4.1 End of Definitions

The lexicon section is terminated by

```
%eolex
```

Kapitel 6

Grammatik

Der im *Semf*-System behandelte Sprachausschnitt (inklusive der Fehlerphänomene) wird durch die in der Grammatik enthaltenen Regeln beschrieben. Zusammen mit dem Lexikon (Kapitel 7) bildet die Grammatik die linguistische Wissensbasis für die Analyse der Lernereingaben. Im Kontext eines Sprachlehrsystems kann davon ausgegangen werden, daß die Benutzerinnen nicht über die volle Grammatik der Zielsprache verfügen. Auf der einen Seite sind also Einschränkungen im Umfang der Grammatik möglich, andererseits sollte das System adäquat mit fehlerhaften Sätzen umgehen können.

Bei der Implementation der *Semf*-Grammatik waren zwei Kriterien zentral:

- Die Grammatik sollte ein hinreichend großes Fragment der einfacheren Konstruktionen des Englischen abdecken. Zusätzlich waren wichtige Fehlerklassen exemplarisch durch die Aufnahme von Fehlerregeln in die Grammatik zu behandeln.[1]
- Die Menge der sprachlich behandelbaren Sätze sollte in etwa mit den Aussagen übereinstimmen, die die Inferenzmaschine (vgl. Kapitel 9) in unserer Beispielwelt beweisen kann. Dieser Punkt ist auch hinsichtlich der Semantikkonstruktion relevant (Kapitel 6.2).

Grammatik und Lexikon werden in *Semf* in eine gemeinsame Rahmen-Datei *grambone.y* eingebunden, deren Struktur in Kapitel 5 erläutert wird. Die tatsächlichen Wissensbasen sind auf die Dateien *grammar.y* (Grammatik), *errorgrammar.y* (Fehlergrammatik) und *lex.y* (Lexikon) verteilt. Die vorliegende Version der Grammatik/Fehlergrammatik enthält 43 Grammatik- und 28 Fehlerregeln. Nach Durchlaufen des *ppp* kommen weitere 79 generierte Fehlerregeln als Varianten der existierenden Regeln hinzu.

In diesem Kapitel soll die implementierte Grammatik dokumentiert werden, zunächst ihr Aufbau und die für die Analyse benutzten syntaktischen Merkmale, dann die Erstellung einer semantischen Repräsentation als Grundlage weiterer Verarbeitung.

[1] Da die Fehlerbehandlung – soweit sie in Grammatik und Lexikon erfolgt – in Kapitel 11.2 detailliert beschrieben ist, werden an dieser Stelle Fehlerregeln weitestgehend ausgespart.

6.1 Syntax
Henrik Dittmann

Dieser Abschnitt beschreibt den Umfang der in *Semf* benutzten Grammatik: Zunächst wird der Aufbau der phrasalen Kategorien erläutert, es folgt eine Übersicht über diejenigen Merkmale, die nicht schon in Kapitel 7 erklärt werden. Abschließend soll der Aufbau einer Grammatikregel anhand eines Beispiels demonstriert werden.

Weitere für das Verständnis der Grammatik wichtige Informationen finden sich in den Kapiteln 5 (Formalismus), 7 (Lexikon, Erklärung einiger Kategorien und Merkmale), 6.2.2 (`SEM`-Merkmalsstruktur) und 11.2.4 (`ERROR`-Merkmalsstruktur).

6.1.1 Aufbau der phrasalen Kategorien

Die folgenden Ausführungen sind nicht vollständig; sie sollen lediglich einen Überblick über den Umfang der verwendeten Grammatik geben. Wir haben die zwischen der Ebene der lexikalischen Kategorien und der Phrasenebene angesiedelten Kategorien nach der Konvention der *X-Bar-Theorie* benannt, über die notationelle Ebene hinaus besteht jedoch keine Anlehnung an diese Theorie.

Der Satz

Bei Analyse der Lernereingabe wird erwartet, daß sie einen vollständigen Satz enthält. Sollten in einer Beispielübung Fragen an die Lernerin gestellt werden, müßte die Grammatik entsprechend um elliptische Formen wie etwa Nominalphrasen oder Präpositionalphrasen erweitert werden.

Die im *Semf*-System analysierbaren Sätze beschränken sich auf Aussagen. Neben Sätzen in der Normalform $S \rightarrow NP\ VP$ werden Sätze behandelt, die Existenz ausdrücken, wie etwa *There is a yellow object behind the large pyramid.* Die einzig behandelte Form von Nebensätzen sind Partizipkonstruktionen wie etwa in *The cube standing on the floor is green.*

Kategorien: `S`

Die Nominalphrase

Ein Nomen kann durch beliebige Adjektiv- (*blue ball*) und Präpositionalphrasen (*thing near the large cube*) zu einer `NBAR`-Konstituente erweitert werden. Außerdem darf eine Partizipkonstruktion vorkommen: *small yellow pyramid behind the blue block standing on the floor.* Eine `NBAR` ergibt zusammen mit einem Determiner[2] – oder im Falle einer indefiniten Plural-

[2]Zu den Determinern zählen die lexikalischen Kategorien `ART`, `DEM`, `NUMER` und `QUANT`.

`NP` auch ohne – eine Nominalphrase. Dabei ist im Falle des unbestimmten Artikels darauf zu achten, daß dieser in Abhängigkeit vom folgenden Phonem *a* oder *an* lauten kann.

Alternativ dazu kann eine `NP` aus einem Personalpronomen bestehen oder auch eine Koordination mehrerer Nominalphrasen enthalten. Für die Implementation der Koordination enthält die Grammatik drei Regeln, die jeweils unterschiedliche Fälle behandeln, abhängig davon, ob die erste Nominalphrase *kopflos* oder die zweite ihrerseits eine koordinierte `NP` ist. Falls die erste Nominalphrase kein Nomen enthält, wird dies als ein elliptischer Ausdruck gewertet, der auf der syntaktischen Seite den Numerus und auf der semantischen die Klassenzugehörigkeit von der zweiten `NP` erbt. Die Funktionalität des dabei verwendeten `STORE`-Merkmals wird weiter unten erklärt.

Die Koordination von Nominalphrasen wird in der *Semf*-Grammatik grundsätzlich als pluralbildend aufgefaßt, was für eine disjuktive Verknüpfung sicherlich nicht immer zutreffend ist. Hier wäre die Verwendung von *funktionalen Ausdrücken* angebracht.

Kategorien: `NP`, `NBAR`, `DET`

Die Verbalphrase

In der *Semf*-Grammatik sind drei Subkategorien von Verben vorgesehen:

Kopula-Verben verlangen als Komplement ein Prädikatsnomen oder Prädikatsadjektiv; auch die Komplemente der präpositionalen Verben sind zulässig.

Transitive Verben verlangen ein direktes Objekt im Akkusativ.

Präpositionale Verben fordern als Ergänzung eine Präpositionalphrase (z.B. *is standing in front of the block*) oder das Adverb *somewhere*.

Das Verb selber kann der Form nach *simple* oder *continuous* sein (gebildet durch eine Form von *to be* und Partizip Präsens). Das Verb, bei dem auf diese Weise der Aspekt festgelegt wurde, bezeichnen wir als `VBAR`-Konstituente. Eine `VBAR` zusammen mit ihrem Komplement bildet die Verbalphrase.

Kategorien: `VP`, `VPT`, `VBAR`

Die Präpositionalphrase

Eine Präpositionalphrase besteht immer aus einer Präposition und einer Nominalphrase im Akkusativ. Im Fall der Präposition *between* wird verlangt, daß die `NP` im Plural steht.

Komplexe Präpositionen wie *in front of* werden als feststehende Lexeme betrachtet. Da jedoch Lexikoneinträge keine Leerzeichen enthalten dürfen, haben wir uns entschlossen, diese im Rahmen der Grammatik zu behandeln. Die gleiche Funktionalität wäre sicherlich auch eleganter erreichbar; da die syntaktische Behandlung von Präpositionalphrasen jedoch keinen Schwerpunkt des Projektes darstellt, entschlossen wir uns für die gegenwärtige Alternative, die eine `PG`-Kategorie als Zwischenstufe annimmt.

Kategorien: `PP`, `PG`

Die Adjektivphrase

Adjektivphrasen bestehen im allgemeinen nur aus einem Adjektiv. Prädikativ gebrauchte Adjektivphrasen, die ein Adjektiv im Komparativ beinhalten, können darüberhinaus jedoch auch eine `NP` mit einem Vergleichsobjekt beinhalten: *bigger than the green block*. In diesem Fall setzt sich die `AP` aus einer `ABAR`-Konstituente, die das Adjektiv und die Konjunktion *than* umfaßt, sowie der `NP` zusammen.

Komplexe Adjektive (z.B. *light blue*) werden genauso behandelt wie komplexe Präpositionen.

Kategorien: `AP`, `ABAR`, `AG`

Die Kategorien `PRO1`, `CONJ1` und `AUX1` sind aus technischen Gründen hinzugefügt worden und unterscheiden sich nicht von den entsprechenden lexikalischen Kategorien `PRO`, `CONJ` und `AUX` (siehe Kapitel 7).

6.1.2 Syntaktische Merkmale

Syntaktische Merkmale sind im Gegensatz zu den in den `SEM`- und `ERROR`-Merkmalsstrukturen enthaltenen Informationen für die weitere Verarbeitung ohne Belang. Sie dienen einzig dazu, Regelanwendungen zuzulassen bzw. sie zu verhindern, wenn bestimmte Kongruenz- oder Rektionsforderungen nicht erfüllt sind. Auf Satzebene liegt deswegen normalerweise kaum noch syntaktische Information vor.

Die folgende Übersicht zählt diejenigen Merkmale in den `SYN`- und `STORE`-Merkmalsstrukturen auf, die nicht in Lexikoneinträgen, sondern erst im Rahmen der syntaktischen Analyse auftreten. Im Wesentlichen handelt es sich hierbei um Merkmale, die als Marker dienen, um in unserer Implementation die Zahl der syntaktischen Kategorien klein zu halten.

`SYN`: *Merkmalsstruktur*
enthält alle für die syntaktische Verarbeitung einer Lernereingabe notwendigen Merkmale.

`STORE`: *Merkmalsstruktur*
speichert einige Informationen syntaktischer wie semantischer Art, die bei der `NP`-Koordination benötigt werden, aber in den der Regel zugänglichen Konstituenten

nicht vorliegen. Außer in den Regeln für die Koordination kommt diese Merkmalsstruktur nirgendwo vor.

`PPEXP: YES, NO`
gibt an, ob eine `NBAR`-Konstituente bereits um eine Präpositionalphrase erweitert wurde. Dieses Merkmal dient dazu, um bei der Erweiterung von Nomina eine Reihenfolge vorzugeben, die verhindert, daß z.B. ein mit einer Adjektiv- und einer Präpositionalphrase erweitertes Nomen zu zwei Parse-Alternativen führt, je nachdem, welche Erweiterung zuerst vorgenommen worden ist.

`VPEXP: YES, NO`
ist auf `YES` gesetzt, wenn eine `NBAR` bereits eine Partizipialkonstruktion enthält. Durch Überprüfung dieses Merkmals wird gewährleistet, daß in jeder `NP` nur maximal eine solche Konstruktion vorkommt.

`PRON: YES, NO`
gibt an, ob eine `NP` aus einem Pronomen besteht.

`HEADLESS: YES, NO`
zeigt an, ob eine Nominalphrase keinen Kopf enthält. Diese Information wird bei der `NP`-Koordination benötigt: *the yellow and the purple thing.*

`NPCOMPLEX: YES, NO`
ist dann gesetzt, wenn eine `NP` aus koordinierten Nominalphrasen besteht.

`ASPECT: SIMPLE, CONTINUOUS`
bezeichnet den Aspekt einer `VBAR`-Konstituente.

`COMPARED_OBJECT_AVAILABLE: YES, NO`
gibt an, ob in einer Adjektivphrase ein Vergleichsobjekt enthalten ist. Dieses Merkmal dient dazu, solche Kostruktionen bei attributiv gebrauchten Adjektiven auszuschließen wie etwa **the bigger than the small block cube.*

6.1.3 Beispiel einer Grammatikregel

Die Struktur einer Grammatikregel und insbesondere die Verwendung von syntaktischen Merkmalen sollen abschließend in einer Beispielregel gezeigt werden. Die folgende Regel beschreibt die Bildung einer `NP` aus einer `NBAR`-Konstituente. Dies entspricht dem Hinzutreten des leeren Artikels für den indefiniten Plural. Die Regel enthält einen Marker (`@ AGR`) für die Generierung einer Regelvariante beim Durchlauf des *ppp* (s. Kapitel 5.1); auf in dieser Variante enthaltene Abweichungen wird im Anschluß kurz eingegangen.

```
NP : NBAR $$- * { *checkEverything } $$+ AGR { #+CONST1 } [
     $0:[
         SYN = [
                CASE = $1:SYN->CASE;
                HEADLESS = $1:SYN->HEADLESS;
                NPCOMPLEX = NO;
                NUM = PLUR;
                ]
         SEM = [
                OBJECTINFO = [
                              QUANTIFIER = [ ... ]
                ]             ]
         SEM = $1:SEM;
         $$+ AGR { ERROR = $1:ERROR; }
         $$- AGR {
           ERROR = [
                    ERR_1 = $1:ERROR;
                    COUNT = 1;
                    COUNT = $1:ERROR->COUNT;
                    ...
                    ] }
         ]
     $1:[
         @ AGR {
           SYN = [
                  NUM = PLUR;
                  ] }
     ]   ]
```

Für die Anwendung der Regel ist es notwendig, daß die NBAR-Konstituente im Plural steht, da sonsten ein Determiner notwendig wäre. Diese Bedingung wird im Abschnitt $1 überprüft. Die syntaktischen Eigenschaften der neuen Konstituente NP sind in der SYN-Merkmalsstruktur im Abschnitt $0 enthalten: Der Numerus ist natürlich Plural, der Kasus und die Information, ob es sich um eine *kopflose* Konstituente handelt, können von der NBAR-Konstituente übernommen werden. Da sie nicht aus einer Koordination hervorgeht, tr"agt diese Nominalphrase das Merkmal NPCOMPLEX = NO.

Zur Semantik der NBAR wird Information über den indefiniten Plural hinzugefügt[3]. Es ist dabei zu beachten, daß die Zuweisung von Werten vor der Unifikation mit anderen Merkmalen zu geschehen hat. Das *subfeature* ERROR ist in dieser Variante (@ AGR enthalten) mit dem der NBAR koreferent.

[3]Diese DET-Information ist für die anderen Determiner in deren Lexikoneinträgen enthalten.

Nur in dieser Variante enthält die Regel in ihrer Kopfzeile außerdem einen Marker (`+CONST1`), der im Fall ihrer Anwendung bewirkt, daß eine Satzkonstruktions-Fehlerregel mit dem gleichen Marker für die Analyse des Satzes an dieser Stelle nicht mehr angewandt wird.

Die *Regelvariante*, in der der mit `@ AGR` markierte Abschnitt entfällt, beschreibt einen Agreementfehler. Bei der Generierung der Variante wird eine Abhängigkeit geschaffen, die gewährleistet, daß bei korrekter Anwendung der eigentlichen Regel die Variante nicht für die Analyse in Betracht gezogen wird. Sie kommt also nur zur Anwendung, wenn die `NBAR` *nicht* im Plural steht.

Das Weglassen der Plural-Bedingung führt zu drei weiteren Änderungen (mit `$$-` markiert) gegenüber der Ausgangsregel:

- In der `ERROR`-Merkmalsstruktur wird eine Fehlerbeschreibung (s. Kapitel 11.2.4) eingefügt.
- Die Kopfzeile enthält einen Aufruf der Funktion *checkEverything*, die in jeder Fehlerregel aufgerufen wird.
- Die Verbindung mit der Satzkonstruktions-Fehlerregel entfällt, da es sich hierbei (fehlender Determiner) um eine konkurrierende Fehlererklärung handelt. Insofern kommt es im Fehlerfall zu mehreren Parse-Alternativen.

6.2 Semantik

In diesem Kapitel sollen die in der Grammatik enthaltenen semantischen Informationen beschrieben werden. Dies betrifft den kompositionellen Teil der Semantik, d.h. die Information darüber, was man aus dem semantischen Wissen von einzelnen grammatikalischen Kategorien über das semantische Wissen einer daraus abgeleiteten Kategorie ableiten kann. Zunächst soll hierfür die Form einer kompositionellen Semantik motiviert werden. Im Anschluß daran wird dann die Funktionsweise der SemF-Semantik erläutert.

6.2.1 Motivation
Klaus Dalinghaus

Zu Beginn der Planung wählten wir als Vorbild die Struktur der kompositionellen Semantik von James Allen. Dieser Repräsentation in einer logischen Form (LF) liegt im Wesentlichen ein 4-Tupel mit folgenden Elementen zugrunde:

- Der **Operator** beschreibt den Typ der Struktur. Dies kann auf Satzebene *PAST* für eine Handlung in der Vergangenheit oder etwa auf der NP-Ebene *DEF/SING* für eine definite Singular-NP sein.

- Der **Name** wird zur eindeutigen Identifizierung verwendet, um Referenzen darzustellen.
- Der **Typ** ist eine Klassifizierung der Struktur. Dies kann z.B. in einer NP *BOY* für einen Jungen sein.
- Der **Modifizierer** ist eine Liste von LF-Strukturen, die die Struktur näher bestimmen.

Beispiele:

1. Die Repräsentation für den Satz "Jill kissed the large boy." könnte lauten:

   (PAST s1 KISS-ACTION (AGENT s1 (NAME j1 PERSON "Jill"))
   (THEME s1 (DEF/SING b1 BOY (LARGE b1))))

   Es handelt sich hier also um eine Handlung in der Vergangenheit vom Typ *KISS-ACTION*, die als Agenten die Person "Jill" und als Thema eine definit Singular klassifizierte Gruppe[4] vom Typ *BOY* mit der näheren Bestimmung *LARGE* hat.

2. Die Repräsentation für den Satz "The small balls lie on a yellow block." könnte lauten:

   (PRES s2 LIE-STATE (AGENT s2 (DEF/PLUR b2 BALL (SMALL b2)))
   (ON-LOC l1 (INDEF/SING b3 BLOCK (YELLOW b3))))

   Es handelt sich hier also um eine Zustandsbeschreibung in der Gegenwart vom Typ *LIE-STATE*, die als Agenten ein Objekt vom Typ *BALL* mit der näheren Bestimmung *SMALL* und als Ort, auf dem es sich befindet, eine indefinit Plural klassifizierte Gruppe vom Typ *BLOCK* mit der näheren Bestimmung *YELLOW* hat.

Wir haben bei der Festlegung des Formalismus für unser System einiges übernommen, aber auch einiges, was uns bei unserer begrenzten Welt nicht relevant erschien, weggelassen.

Als Struktur haben wir die Feature-Term-Struktur gewählt, um bei der kompositionellen Semantik das gleiche Format wie bei der Syntax zu haben. Daraus ergaben sich einige Änderungen. Aus Operatoren wie *AGENT* wurden Feature-Namen wie *OBJECTINFO*[5]. Andere Operatoren lösten wir komplex auf, so wurde z.B. aus *ON-LOC* das komplexe Feature *OPERAND*, in dem als Wert *RELATION=ON;* und die restlichen Informationen

[4] also ein Objekt im weitesten Sinne

[5] Der Name erklärt sich durch die Tatsache, daß wir Sätze behandeln wollen, die eine Objektwelt beschreiben

über den Ort[6] enthalten sind. Wieder andere Operatoren, die keinen Typ zur näheren Bestimmung haben wie *SMALL* oder *YELLOW*, lösten wir zu Werten von *ATTRIBUTE* auf.

Beispiel: Die Repräsentation für den Satz "The small balls lie on a yellow block." lautet in Semf:

$$
SEM : \begin{bmatrix}
\text{OBJECTINFO:} & \begin{bmatrix} \text{CLASS:} & BALL \\ \text{ATTRIBUTE:} & SMALL \\ \text{ORIENTATION:} & LIES \\ \text{QUANTIFIER:} & \begin{bmatrix} \text{LABEL:} & DET\_THE \\ \text{RANGE:} & 2\_\%100 \\ \text{ANAPHORA:} & POSSIBLE \\ \text{AMBIGUITY:} & YES \end{bmatrix} \\ \text{COMPARE:} & POSI \end{bmatrix} \\
\text{OPERAND:} & \begin{bmatrix} \text{RELATION:} & ON \\ \text{OBJECTINFO:} & \begin{bmatrix} \text{CLASS:} & BLOCK \\ \text{ATTRIBUTE:} & YELLOW \\ \text{QUANTIFIER:} & \boxed{1} \end{bmatrix} \end{bmatrix}
\end{bmatrix}
$$

$$
\boxed{1} = \begin{bmatrix} \text{LABEL:} & DET\_THE \\ \text{RANGE:} & 1\_\%100 \\ \text{ANAPHORA:} & POSSIBLE \\ \text{AMBIGUITY:} & YES \end{bmatrix}
$$

Die eindeutigen Namen für einzelne Operatoren sind nicht mehr im Formalismus enthalten. Die Liste der Modifizierer ist als zusätzliche Information in weiteren Features enthalten. Weiter wurde die Struktur durch das Weglassen von verschiedenen Operatoren wie etwa dem einleitenden Operator zur Bestimmung der Satzart vereinfacht. Da wir davon ausgehen, nur mit Objekten zu operieren, können wir uns bei der Formalisierung einer Nominalphrase auf ein Feature *CLASS* beschränken. Der Quantor wird in einem weiteren Feature zusätzlich behandelt.

Die Produktion dieser semantischen Strukturen erfolgt über den Weg der Unifikation von Feature-Term-Strukturen. über diesen Weg ist es in jeder Regel der Grammatik möglich, aus den semantischen Repräsentationen der einzelnen Satzteile die Repräsentation für den neuen Satzteil zu bilden.

Durch diese massiven Einschränkungen hat der Formalismus erheblich an Ausdrucksstärke verloren. So sind Referenzen zwischen einzelnen Objekten nicht mehr möglich, d.h. Pronomina können in diesem Formalismus nicht den zugehörigen Nominalphrasen zugeordnet werden. Weiter vermindert sich durch das Mischen von Größen, Farben, etc. unter dem Feature-Namen *ATTRIBUTE* der Informationsgehalt in der semantischen Repräsentation, da man nur noch weiß, daß ein Objekt das Attribut *LARGE* hat, aber nicht mehr

[6] diese sind wiederum als *OBJECTINFO* vorhanden

weiß, daß es sich dabei um ein Attribut der Klasse Größe handelt.

Viele dieser Phänomene kann man jedoch auf der rein syntaktischen Ebene nicht lösen. So kann man bei der syntaktischen Analyse eines Satzes nicht sagen, ob das Wort "green" nun ein Attribut der Klasse Farbe oder der Klasse Wissensstand[7] repräsentiert. Auch bei Pronomina kann man auf der rein syntaktischen Ebene nicht festlegen, auf welche Nominalphrase sie sich nun beziehen.

Da die Informationen später jedoch in einer semantischen Komponente weiter verarbeitet werden, und dort eine Behandlung solcher Probleme möglich wird, geht somit dem Gesamtsystem nichts an behandelbarer Mächtigkeit verloren. Das bedeutet, der Formalismus ist auf die für den syntaktischen Teil der Analyse relevanten Inhalte der kompositionellen Semantik beschränkt worden. Das Ergebnis wird anschließend in einer Komponente "Nichtkompositionelle Semantik" weiter aufbereitet.

Wichtige positive Eigenschaften dieser Struktur blieben jedoch erhalten. Hier ist vor allem die Rekursivität zu erwähnen. Innerhalb der Beschreibung einer NP kann also durchaus die Beschreibung einer NP vorkommen, wodurch beliebige Schachtelungen möglich werden. Hinzugefügt wurde ein Mechanismus zur Behandlung von Listen, d.h. zu einem bestimmten Feature (z.B. *OBJECTINFO*) gibt es ein Feature *NEXT*, in dem es wieder dieses Feature gibt und ggf. noch ein Feature *NEXT*.

6.2.2 Funktionalität
Henrik Dittmann

Im Folgenden soll die Implementierung der kompositionellen Satzsemantik im *Semf*-System beschrieben werden, die während des Parsings des Eingabesatzes erstellt wird. Der Schwerpunkt liegt dabei auf der Bildung komplexer semantischer Einträge, auf die verwendeten Merkmale soll nur insoweit eingegangen werden, als dies für das Verständnis notwendig erscheint. Soweit sie nicht explizit für die Kompositionalität wichtig sind, werden die semantischen Merkmale in Kapitel 7.3 näher beschrieben.

Die für die weitere semantische Verarbeitung benötigten Angaben über den Eingabesatz sind im `SEM`-Merkmal jeder Konstituente enthalten. Diese Informationen sind im Sinne einer allgemeinen semantischen Repräsentation unvollständig, im Vordergrund der Semantikkonstruktion stand im *Semf*-System jedoch die leichte Übersetzbarkeit in die Anfragesprache der Inferenzmaschine (vgl. auch Kapitel 8 und 9). Für die Verwendung in anderen, ähnlichen Kontexten kann diese Repräsentation leicht geändert oder ergänzt werden.

Struktur der semantischen Repräsentation

Das `SEM`-Merkmal ist stark verschachtelt und bei komplexeren Konstituenten in der Regel rekursiv aufgebaut. Es ist zunächst in zwei Merkmale vom Typ *subfeature* aufgeteilt:

[7]Beispiel: I'm not as green as I look.

- Das `OBJECTINFO` enthält sämtliche Informationen über das Subjekt, die aus der Subjekts-Nominalphrase und gegebenenfalls auftretendem Prädikativ ("The object is *a ball.*") gewonnen werden.

- Der `OPERAND` beschreibt die Beziehung zu einem anderen Objekt in der Welt, die in der Verbalphrase durch ein transitives Verb oder eine Präpositionalphrase ausgedrückt sein kann.

In dieser Zweiteilung spiegelt sich der objektorientierte Ansatz wider, der in der Weltrepräsentation des *Semf*-Systems verfolgt wird.

Die innere Struktur des `OBJECTINFO`

Ein `OBJECTINFO`-Merkmal enthält im weitesten Sinne die Semantik einer Nominalphrase. Im einfachsten Fall umfaßt das `OBJECTINFO` nur folgende Merkmale:

- das `CLASS`-Merkmal, das semantische Informationen über das Nomen enthält;

- das `QUANTIFIER`-Merkmal, das semantische Informationen über den Artikel enthält. Dieses Merkmal ist ein *subfeature*, das seinerseits die Merkmale `ANAPHORA`, `LABEL` und `RANGE` beinhaltet.

Falls es sich bei der Nominalphrase um ein Pronomen handelt, werden beide Merkmale aus der lexikalischen Semantik des Pronomens übernommen.

Weitere Merkmale, die innerhalb des `OBJECTINFO` auftreten können, sind

- das `ATTRIBUTE`-Merkmal, das semantische Informationen der Adjektivphrase enthält;

- das `ORIENTATION`-Merkmal, das hinzugefügt wird, wenn das Nomen logisches Subjekt eines präpositionalen Verbs (z.B. *to stand*) ist.

Die bisher beschriebenen Merkmale sind alle als *attrlistdef* definiert, so daß z.B. bei mehreren Adjektiven das `ATTRIBUTE`-Merkmal mehrere Werte annimmt (konjunktiv verknüpft).

Neben der so beschriebenen Standard-Nominalphrase existieren jedoch einige Sonderfälle, die Abweichungen bzw. Ergänzungen der `OBJECTINFO`-Struktur erfordern:

Eingebettete Lokalisierungen In einem Satz wie "The ball *behind the large pyramid* is lying on the floor." tritt eine in die Nominalphrase eingebettete Präpositionalphrase oder Partizipialkonstruktion ("the pyramid *standing on the cube*") auf, die (in der räumlichen Welt von *Semf*) den Ort des Objektes näher beschreibt. Diese Information wird in einem `OPERAND`-Merkmal gespeichert, das im `OBJECTINFO` eingebettet ist. Die Struktur dieses komplexen Merkmals wird im folgenden Abschnitt näher beschrieben, allerdings können an dieser Stelle mehrere Lokalisierungen auftreten, so daß eine Listenstruktur erforderlich wird (erzeugt in den Regeln `NBAR -> NBAR PP` und `NBAR -> NBAR VP`).

Ein Beispiel für die Struktur eines solchen komplexen `OPERAND`-Merkmals soll die Nominalphrase *the block next to the pyramid on the left of the cube behind the object lying on the floor* geben; der beschriebene Fall ist der, daß sich alle Lokalisierungen auf *the block* beziehen:

```
OBJECTINFO = [
               ...
               OPERAND = [
                          FIRST = [lying on the floor]
                          NEXT = [
                                  FIRST = [behind the object]
                                  NEXT = [
                                          FIRST = [on the cube]
                                          NEXT = [
                                                  FIRST = [next to the pyramid]
                                                  NEXT = []
               ]          ]       ]       ]       ]
```

Eine Interpretation, die z.B. *lying on the floor* als nähere Bestimmung zu *the object* auffaßt, enthält eine weniger tiefe Liste, dafür allerdings ein komplexeres `OBJECTINFO` als Bestandteil des `FIRST`-Merkmals, das die Lokalisierung *on the cube next to the pyramid* beschreibt (vgl. den Abschnitt über die innere Struktur des `OPERAND`).

Prädikative Ergänzungen Um der Notwendigkeit begegnen zu können, zwischen den Sätzen (1) *All blue balls are big.* und (2) *All big balls are blue.* zu unterscheiden, sieht die *Semf*-Grammatik ein `COMP`-Merkmal vor, das in prädikativen Ergänzungen enthaltene Informationen über die Subjekts-Nominalphrase aufnimmt. Prädikative und attributive Ergänzungen werden also an verschiedenen Stellen im `OBJECTINFO`-Merkmal abgelegt. Die Merkmalsstrukturen der beiden Beispielsätze sehen (auf die relevanten Merkmale beschränkt) so aus:

$$
(1)\quad \begin{bmatrix} \text{OPERAND} & [] \\ \text{OBJECTINFO} & \begin{bmatrix} \ldots & \textit{all balls} \\ \text{ATTRIBUTE} & \text{BLUE} \\ \text{COMP} & \begin{bmatrix} \text{ATTRIBUTE} & \text{LARGE} \end{bmatrix} \end{bmatrix} \end{bmatrix}
$$

$$
(2)\quad \begin{bmatrix} \text{OPERAND} & [] \\ \text{OBJECTINFO} & \begin{bmatrix} \ldots & \textit{all balls} \\ \text{ATTRIBUTE} & \text{LARGE} \\ \text{COMP} & \begin{bmatrix} \text{ATTRIBUTE} & \text{BLUE} \end{bmatrix} \end{bmatrix} \end{bmatrix}
$$

Komparative Adjektive treten in *Semf* in allen Stufen der Komparation auf. Neben dem `ATTRIBUTE`-Merkmal enthalten die im Lexikon enthaltenen Adjektive deshalb auch einen `COMPARE`-Eintrag, der die Komparationsstufe angibt. Falls ein Prädikatsadjektiv im Komparativ steht, wird die semantische Information über das Vergleichsobjekt im Merkmal `COMPARED_OBJECT` abgelegt (Regel `AP -> ABAR NP`). Dieses tritt nur innerhalb des `COMP`-Merkmals auf, da bei attributiv gebrauchten Adjektiven kein Vergleichsobjekt vorkommen kann.

Koordination von Nominalphrasen Die *Semf*-Grammatik läßt nur die Koordination abgeschlossener Nominalphrasen zu - die NP *the yellow and the green pyramid on the floor* etwa hat nur die Lesart, in der *on the floor* sich auf *the green pyramid* bezieht.[8] Diese Koordination äußert sich in der kompositionellen Semantik derart, daß das `OBJECTINFO`-Merkmal als einziges ein `COMPOUND`-Merkmal enthält, das seinerseits für die koordinierten Nominalphrasen zwei *subfeatures* `OBJ1` und `OBJ2` enthält. `OBJ2` kann eine weitere Koordination enthalten.

Die innere Struktur des `OPERAND`

Das `OPERAND`-Merkmal enthält eine Lokalisierung des Subjektes bzw. als in ein `OBJECTINFO` eingebettetes Merkmal auch die von Objekten. Es tritt auf, wenn durch eine Präposition oder ein transitives Verb eine Beziehung zwischen mehreren Objekten in der Welt hergestellt wird und enthält normalerweise zwei Merkmale:

- das `RELATION`-Merkmal, das die durch die Präposition oder das transitive Verb gegebene Relation enthält;[9]
- ein `OBJECTINFO`-Merkmal (vgl. den vorhergehenden Abschnitt), das das Referenzobjekt beschreibt.

Als Lokalisierung kann auch der Eintrag des Adverbs *somewhere* gelten, der statt der beiden Merkmale `RELATION` und `OBJECTINFO` das `ADVERBIAL`-Merkmal einfügt.

Als Beispiel für die gegenseitige Einbettung von `OBJECTINFO`- und `OPERAND`-Merkmalen sei der Satz *The red pyramid touching the small ball stands between the pink and the purple object on the floor.* gegeben. Auf Satzebene hat das `SEM`-Merkmal die folgende Struktur:

---

[8] *the yellow* wird als eine kopflose Nominalphrase behandelt.

[9] Man beachte, daß die präpositionalen Verben in *Semf* keine Relation ausdrücken, sondern den `ORIENTATION`-Eintrag im `OBJECTINFO` füllen.

```
SEM = [
       OBJECTINFO = [
                     ... [the red pyramid]
                     ORIENTATION = STANDS;
                     OPERAND = [
                                RELATION = TOUCH;
                                OBJECTINFO = [the small ball]
                     ]          ]
       OPERAND = [
                  RELATION = BETWEEN;
                  OBJECTINFO = [
                    COMPOUND = [
                                OBJ1 = [the pink object]
                                OBJ2 = [
                                        ... [the purple object]
                                        OPERAND = [
                                                   RELATION = ON;
                                                   OBJECTINFO = [the floor]
       ]          ]            ]]       ]          ]
```

Kapitel 7

Lexikon
Alexander Krumeich

7.1 Format des Lexikons

Im Rahmen der Entwicklung von Semf erstellten wir ein Vollformenlexikon, das im wesentlichen das Sprachfragment abdeckt, das wir brauchen, um unsere Blockswelt sprachlich wiedergeben zu können. In bezug auf das Lexikon ist das Semf System jedoch sehr flexibel. Dem Programmierer bleibt es zum einen vorbehalten, ein eigenes Lexikon zur Verfügung zu stellen, zum anderen kann er auch ein „externes" Lexikon verwenden. Die Einbindung des Lexikons erfolgt über eine fest definierte Schnittstelle zum Parser.

Das Lexikon enthält 113 Einträge in 12 Hauptkategorien. Es gibt noch einige weitere (Hilfs-)Kategorien, die aber im wesentlichen Spezialfälle darstellen und deren Mitglieder leicht einer anderen linguistischen Kategorie zugeordnet werden können.

Das Lexikonformat ist nah an das der Grammatik angelehnt. Ähnlich wie bei den Regeln gibt es verschiedene Subfeatures (`SYN`, `SEM` und `ERROR`), die im Rahmen der Unifikation an höhere Kategorien weitergereicht werden. Anders jedoch als in der Grammatik hat ein Lexikoneintrag keine Unterscheidung zwischen `$0:` und `$n` Features. Lexikoneinträge haben in der Regel folgendes Format:

```
lexical_item    CAT [
                     SYN = [
                             feature 1 = value 1;
                             feature n = value n;
                            ]
                     SEM = [
                             feature 1 = value 1;
                             feature n = value n;
                            ]
                   ERROR = [
                             feature 1 = value 1;
                             feature n = value n;
                            ]
                ]
```

Hierbei ist `lexical_item` das Lexem und `CAT` die Kategorie, der das Lexem zugeordnet wird. Folgende Hauptkategorien sind in Verwendung[1]:

[1] Die Hilfskategorien sind hier nicht aufgeführt, deren Einträge sind den linguistisch motivierteren Kategorien zugerechnet

CAT	Kategorie	Anzahl der Einträge
PRO	Personalpronomen	4
ART	Artikel	4
DEM	Demonstrativpronomen	2
NUMER	Numeral	10
QUANT	Quantoren	5
N	Nomen	17
P	Präposition	20
A	Adjektiv	31
V	Verb	15
AUX	Auxiliarverb	2
ADV	Adverb	1
CONJ	Konjunktion	2

SYN, SEM und ERROR sind die drei verwendeten Subfeatures. SYN enthält syntaktische Information, in SEM stehen Angaben über semantische Information und in ERROR befinden sich Angaben über den lexikalischen Fehler. Das ERROR-Subfeature taucht natürlich nur bei Einträgen im Fehlerlexikon auf. Im folgenden werden die möglichen Features und Werte erläutert.

7.2 Das SYN-Feature

Alle hier angegebenen Attribute sind „normale“ Attribute, in dem Sinne, daß immer nur einer der möglichen Werte vergeben werden kann.

CASE: NOM, ACC
: Kasus. Entweder Nominativ oder Akkusativ

NUM: SING, PLUR
: Numerus. Entweder Singular oder Plural

FORM: FINITE, PRESPART
: Form des Verbs. Die -ing Form eines Verbs erhält den Wert PRESPART, die Simple Present Form den Wert FINITE. Nur Verbalphrasen mit dem Wert FINITE werden zur Satzbildung zugelassen. Dadurch werden Sätze wie **The block supporting the cube* ausgeschlossen.

SUBCAT: INTENSIVE, TRANSITIVE, PREPOSITIONAL
: Die Subkategorisierungsliste eines Verbs. Der Wert INTENSIVE wird vergeben, wenn ein Prädikatsnomen subkategorisiert werden soll (wie bei *to be*). Ein mit dem Wert TRANSITIVE versehener Eintrag fordert ein Akkusativobjekt als Ergänzung, PREPOSITIONAL fordert eine PP.

CONJ_TYPE: COORD, OTHER
: Typ der Konjunktion. Dient zur Unterscheidung der verschiedenen Konjunktionen. Koordinierende Konjunktionen (*and*) erhalten als Wert COORD, andere (*than*) den Wert OTHER.

Damit wird die Bildung von **The Block than the cube are red* oder **The Block is bigger and the cube* verhindert.

`INITIAL`: `CONSONANT`, `VOWEL`
: Stellt die *a/an* Kongruenz zwischen Nomen und Artikeln her. Nur wenn sowohl Nomen als auch Artikel identische Werte in diesem Feature haben, können sie zusammen eine NP bilden.

`UNIQUE`: `YES`, `NO`
: Gibt an, ob zu einem Lexem mehrere Einträge im Lexikon vorliegen.

`THERE`: `YES`, `NO`
: Ein Hilfsfeature für den Eintrag *there*. Um Sätze mit dem pronominalen Dummy-Subjekt verarbeiten zu können. Auf der anderen Seite soll verhindert werden, daß *there* an Stellen verwendet wird, wo es zu einem inkorrektem Parse führt: **There are green.*

7.3 Das SEM-Feature

Während sich das `SYN`-Feature als eine flache Struktur darstellt, kommt es im `SEM`-Feature immer zu Verschachtelungen. Das Subfeature `OBJECTINFO` enthält Information über ein Objekt. Im Semf Kontext sprechen wir von der kompositionellen Semantik. Dieses Subfeature ist bei fast allen Kategorien zu finden. Im `OBJECTINFO` einer Kategorie auf einer hohen Ebene (z.B. NP) finden sich nach der Unifikation zum Teil sehr umfangreiche und präzise Beschreibungen eines Objekts. Das `OBJECTINFO`-Subfeature ist oftmals selbst verschachtelt. So enthält es in den meisten Fällen ein Subfeature `QUANTIFIER`, in dem der verwendete Quantor genauer spezifiziert wird.

Ein weiteres Subfeature ist `OPERAND`. Als Operand betrachtet das Semf-System PPn oder subkategorisierte NPn. Der zugehörige Operator ist demnach eine Präposition oder ein transitives Verb. `OPERAND` enthält wiederum ein `OBJECTINFO`-Subfeature, in dem der Operand genauer beschrieben wird. Zusätzlich verfügt `OPERAND` noch über ein weiteres Feature (`RELATION`), das nähere Angaben über die Art der Präposition macht.

`COMPARE`: `POSI`, `COMPA`, `SUPER`
: Steigerungsgrad eines Adjektivs. Die Werte entsprechen den Formen Positiv, Komparativ und Superlativ. Vom Wert dieses Features hängen die Einbettungsmöglickeiten in ein Adjektivgefüge ab.

`ADVERBIAL`: `SOMEWHERE` Lokalisierung. Wird nur bei *somewhere* gesetzt.

`ANAPHORA`: `YES`, `NO`, `POSSIBLE`
: Legt bei Pronomina und Artikeln fest, ob es sich bei der Konstituente um eine Anapher handelt. Pronomina haben immer den Wert `YES`, indefinite Artikel sind keine Anaphern (`NO`), bei definiten Artikeln ist anaphorischer Gebrauch möglich (`POSSIBLE`).

`LABEL`: `PRO_IT`, `THERE`, `DET_A`, `DET_THE`, `DET_THIS`, `DET_ONE`, `DET_TWO`, `DET_THREE`, `DET_FOUR`, `DET_FIVE`, `DET_SIX`, `DET_SEVEN`, `DET_EIGHT`, `DET_NINE`, `DET_TEN`, `DET_NO`, `DET_SOME`,

DET_MOST, DET_EVERY, DET_ALL
Legt eindeutige Bestimmungsmoeglichkeit fuer die nichtkompositionelle Semantik fest.

ATTRIBUTE: BLACK, BLUE, CYAN, DARKGRAY, GRAY, GREEN, LIGHTGRAY, MAGENTA, ORANGE, PINK, RED, WHITE, YELLOW, BIG, HIGH, LARGE, SMALL
Attribute eines Objekts. Die Attribute sind Eigenschaften der Adjektive. Während der Unifikation werden möglicherweise mehrere Attribute in ein OBJECTINFO hineinunifiziert, so daß sehr gut zu sehen ist, wie sich die kompositionelle Semantik einer komplexen NP langsam aufbaut. Da ein Objekt mehrere Attribute haben kann, ist dieses Feature als ein Listenfeature definiert, d.h. die Unifikation gelingt immer.

CLASS: BALL, BLOCK, CUBE, FLOOR, OBJECT, PYRAMID, ENTITY, SOMEWHERE, PRONOUN, NO_HEAD
Analog zu ATTRIBUTE, das die kompositonelle Semantik Adjektiven enthält, befindet sich im Attribut CLASS Information darüber, welcher Objektklasse ein Nomen angehört. Hier gibt es einige Sonderfälle, die eher technischer Natur sind: SOMEWHERE, PRONOUN, NO_HEAD. Letzteres wird bei elliptischen NP-Koordinationen verwendet, bei denen wir Adjektiven nominalen Status zuweisen (*The red and the blue ball*). Wir bezeichnen solche Nominalphrasen als *kopflos*.

ORIENTATION: LIES, STANDS
Eintrag von Verben, der festlegt, in welcher Relation ein Nomen zu einem anderen steht. Da das Feature als Attributliste definiert ist, können auch beide Werte gleichzeitig angenommen werden.

RELATION: BEHIND, BENEATH, BETWEEN, FRONT, NEXTTO, ON, LEFT, RIGHT, TOUCH
Ein weiteres Listenfeature, das die Relation beschreibt, die durch Präpositionen oder Verben gegeben werden. So werden verschiedene Präpositionen auf identische Raumrelationen abgebildet, z.B. *at* → NEXTTO.

RANGE: <lo>_<hi>
Ein Stringfeature (die möglichen Werte müssen nicht vereinbart werden, jede Kette aus ["%\'\~A-Za-z0-9_] kann hier verwendet werden). RANGE wird bei Quantoren verwendet und enthält den Wert des Quantors, wobei <lo> die niedrige Schwelle und <hi> die höhere Schwelle ist. Es können sowohl absolute Zahlen angegeben werden als auch prozentuale Angaben. Bei letzteren ist dem Zahlenwert ein % voranzustellen.

Kapitel 8

Nichtkompositionelle Semantik
Georg Rehm

8.1 Motivation

Das Semf-System besteht im wesentlichen aus einem Parser und einer Inferenzmaschine. Es muß gewährleistet werden, daß Daten, die zwischen den beiden Moduln fließen, verträglich sind. Das Ausgabeformat des Parsers (Featurestrukturen) ist sehr unterschiedlich von dem der Inferenzmaschine und der Weltrepräsentation, welches listenähnlich aufgebaut ist. Die nichtkompositionelle Semantik des Semf-Systems wandelt die vom Parser ausgegebene semantische Analyse eines Eingabesatzes in ein Format um, das von der Inferenzmaschine weiterverarbeitet werden kann. Im Zuge dieses Prozesses wird auch versucht, eventuell vorhandene Anaphern, z.B. Pronomen, aufzulösen. Verfahren zum Behandeln von Quantorenskopusambiguitäten wurden ansatzweise entwickelt, jedoch nicht implementiert.

8.2 Randbedingungen

Es existieren verschiedene Randbedingungen, die das Konzept der nichtkompositionellen Semantik beeinflußt haben. So verfügen wir nur über ein sehr kleines Fragment des Englischen, da die dem System zugrundeliegende Blockswelt eine sehr grobe Granularität besitzt. Daraus folgt, daß in dem von uns entwickelten Lexikon nur relativ wenige Einträge für Objekte und Eigenschaften dieser Objekte zu finden sind, z.B. *ball*, *block*, *large* oder *blue*.

Unser Augenmerk liegt auf dem Beschreiben von Relationen zwischen Objekten, also Präpositionen. In diesem Zusammenhang wurde zu Beginn des Projekts eine Sammlung von Beispielsätzen (siehe Anhang) erstellt, die unser System zu verarbeiten in der Lage sein sollte. Strukturell sind diese Beispielsätze einander sehr ähnlich: Es werden nahezu immer Relationen zwischen verschiedenen Objekten beschrieben. Daraus ergibt sich, daß die Umschreibeprozedur der nichtkompositionellen Semantik fast ausschließlich Anfragen der Art `? (Ball color=blue 1 1 e|on|Cube size=large 1 1 e)` konstruieren muß.

Des weiteren leistet schon unser Parser sehr viele Aufgaben in Bezug auf die kompositionelle semantische Analyse einer Eingabe, eine weitere Randbedingung. Es werden oft unterschiedliche Satzanalysen konstuiert, die – jede für sich – auf ihren Wahrheitswert in der jeweiligen Blockswelt-konfiguration überprüft werden müssen. Hierbei werden sprachliche Objekte auf Konzeptgruppen und Präpositionen auf Relationen zwischen den Konzeptgruppen abbildet. Eine weitere wichtige Randbedingung war, daß wir nur eine Behandlung von im Kontext der Blockswelt relevanten Sätzen leisten wollten, eine robuste Fehlerbehandlung[1] war an dieser Stelle nicht vorgesehen.

Unsere Vorgehensweise bei der Auflösung von Anaphern ist nicht auf eine der grundlegenden semantischen Theorien, z.B. DRT, zurückzuführen. Vielmehr war unser Ziel, Anaphern wie *it* mit einfachen Mitteln behandeln zu können. Eines der Ziele bei der Konzeptionierung des Semf-Systems war, daß Diskurse, z.B. "*There is a blue ball. It is large.*", möglich sein sollten. Es stellte sich jedoch schon bald heraus, daß eine vollständige und robuste Implementierung einer solchen Komponente innerhalb des vorgesehenen Projektzeitraums nicht geleistet werden kann. Daher haben wir nur einen Teil dieses Moduls beispielhaft implementiert: Die Auflösung von anaphorischen Pronomina wie beispielsweise "it".

[1] Also das Detektieren und Erklären von im Kontext irrelevanten Schülereingaben.

8.3 Umschreibeprozedur

Die Ausgabe des Parsers beinhaltet unter anderem eine Darstellung der kompositionellen Semantik eines Eingabesatzes. Diese kompositionelle Semantik wird nun analysiert und in das Format umgeschrieben, das von der Inferenzmaschine als Anfragesprache verstanden wird. Es folgt eine Beispielausgabe des Parsers für den Satz "*The block is on the floor.*"

```
SEM = [
       OBJECTINFO = [
                      QUANTIFIER = [
                                     ANAPHORA = POSSIBLE;
                                     RANGE = 1_1;
                                    ]
                      CLASS = BLOCK;
                     ];
       OPERAND = [
                   RELATION = ON;
                   OBJECTINFO = [
                                  CLASS = FLOOR;
                                  QUANTIFIER = [
                                                 ANAPHORA = POSSIBLE;
                                                 RANGE = 1_1;
                                                ];
                                 ];
                  ];
      ];
```

Mittels der Technik des rekursiven Abstiegs (recursive descent) wird dieses Format nun in eine gültige Anfrage an die Inferenzmaschine konvertiert:

```
? (Block 1 1 e|on|Floor 1 1 e)
```

Hierbei werden auch einigen Relation umgeschrieben, da manche Benennungen von Relationen zwischen Lexikon und Weltrepräsentation unterschiedlich sind.

8.4 Anaphernresolution

Neben einer generellen Aufforderung, eine bestimmte Blocksweltkonfiguration zu beschreiben, sind auch Aufgaben der Art "*Where is the red block?*" vorstellbar. Es ist sehr wahrscheinlich, daß ein Schüler auf eine solche Frage mit einer pronominalen Konstruktion wie z.B. "*It is behind the large pyramid.*" antwortet. Im Rahmen eines wissensbasierten Systems wie Semf ist es sehr wichtig, den korrekten Koreferenten für das Pronomen "*it*" aufzulösen, damit die vom Schüler intendierten Objekte für den Vergleich mit der Weltrepräsentation herangezogen werden können.

Ein weiteres Szenario ist denkbar: Der Schüler einen Satz ein wie "*The red block is on the floor and the blue cube is large.*" Führt er seine Beschreibung mit der Aussage "*The yellow one is small.*" fort, so tritt eine Ambiguität auf, da nicht evident ist, ob sich *the yellow one* auf einen *block* oder einen *cube* bezieht. Auch in diesem Fall ist wünschenswert, zwischen den beiden Lesarten unterscheiden zu können.

Ambiguitäten treten auch im folgenden Beispiel auf: "*A ball is on the cube. It is red.*" Der Schüler bezieht sich hier mit einer anaphorischen Konstruktion auf eine Nominalphrase, die von ihm selbst im vorigen Satz eingeführt wurde. Das Pronomen *it* fungiert hier als eindeutiger Marker für eine anaphorische Konstruktion. Jedoch ist zunächst unklar, ob sich der Schüler auf den *ball* oder den *cube* bezieht. Da der Ball im ersten Satz an der Subjektsposition steht, kann man annehmen, daß der Schüler versuchte, mit dem zweiten Satz eben diesen Ball näher zu beschreiben. Annotiert man Nominalphrasen beim Parsen mit einem zusätzlichen Feature `NP=SUB` bzw. `NP=OBJ`, ist es möglich, Subjekts-Nominalphrasen bevorzugt als Koreferenten zu anaphorischen Konstruktionen aufzulösen.

Das folgende Beispiel motiviert, warum wir im Rahmen der Anaphernauflösung auch Informationen über die Menge der Objekte benötigen, die – entweder vom Schüler oder vom System – eingeführt werden: "*There are green blocks. They are large.*" Zunächst macht der Schüler eine Aussage über die Existenz von grünen Blöcken. Daraufhin behauptet er, sie seien auch groß. Diese Aussage muß für jeden grünen Block der aktuellen Blocksweltkonfiguration gelten, damit sie wahr ist.

8.4.1 Anapherndetektion

Anaphern sind im Kontext des Semf-Systems daran zu erkennen, daß eines der folgenden Kennwörter bzw. eine der folgenden Phrasen auftritt. Dabei sind verschiedene Abstufungen zu unterscheiden.

- Personalpronomen: *he, she, it, they, them, this* [*object*]
  Taucht eines dieser Pronomen auf, so fügt der Parser das Feature `ANAPHORA=YES` an die entsprechende Phrase an. Dieses Merkmal wird im Zuge der Anaphernauflösung von der nichtkompositionellen Semantik erkannt.

- Definite Nominalphrasen: *the ball, the pyramids, the blue balls, the red one, ...*
  Bei diesen entweder im Singular oder im Plural stehenden definiten Nominalphrasen kann es sich unter Umständen um einen anaphorischen Bezug handeln. Solche Phrasen werden ebenfalls mit dem Anapher-Feature, jedoch mit einem anderen Wert annotiert: `ANAPHORA=POSSIBLE`.

- Sonstige: Indefinite Nominalphrasen, Subjekt und Objekt des ersten Satzes einer Übung sowie einige Sonderfälle erhalten das Feature `ANAPHORA=NO`.

8.4.2 Stack

Der Lösungsansatz für die Anaphernresolution ist der Einsatz eines Stacks, auf dem annotierte Nominalphrasen abgelegt werden. Taucht eine Anapher auf, wird untersucht, ob sie sich auf ein Objekt, das auf dem Stack liegt, bezieht. Wenn ja, so wird die eindeutige Nummer des auf dem Stack befindlichen Objekts herangezogen, um einen Vergleich mit der Welt durch die Inferenzmaschine zu ermöglichen. Handelt es sich jedoch nicht um einen anaphorischen Bezug, so wird das Objekt, das in der semantischen Struktur mit dem Merkmal `ANAPHORA=POSSIBLE` annotiert ist, als neues Objekt auf den Stack gelegt, da sich der Schüler ja in seiner nächsten Aussage auf genau dieses Objekt anaphorisch beziehen kann. Die beiden folgenden Beispielsätze motivieren den Einsatz des Stacks sowie der genannten Zusatzmarkierungen.

Satz I

Der Schüler gibt ein: "*There is a green ball.*" Der Parser erkennt daraufhin, daß es sich um eine indefinite Subjekts-Nominalphrase handelt; es liegt also kein anaphorischer Bezug vor, und der Range dieser Phrase liegt zwischen 1 und 1. Die nichtkompositionelle Semantik erkennt daraufhin, daß es sich nicht um eine Anapher handelt, da die Phrase vom Parser mit dem Merkmal `ANAPHORA=NO` annotiert wurde. Das Objekt mit den Eigenschaften *ball, green, SUB, 1_1* wird nun auf den Stack gelegt, um evtl. später darauf referieren zu können. Nun wird die Anfrage an die Inferenzmaschine konstruiert. Diese liefert unter anderem die Information, daß die eindeutige Nummer des Objekts *12* ist. Diese Nummer wird nun ebenfalls in den Stack eingetragen, damit dieser Berechnungsschritt nicht erneut erfolgen muß.

Satz II

Der Schüler gibt ein: "*It is huge.*" Der Parser erkennt, daß *it* eine Anapher ist, weil es sich um ein Pronomen handelt und annotiert die Subjekts-Nominalphrase mit dem Merkmal `ANAPHORA=YES`. Die nichtkompositionelle Semantik erkennt dieses Feature und konsultiert daraufhin den Stack, der als oberstes Objekt *ball, green, SUB, 1_1, 12* zurückliefert. Diese Daten werden nun mit dem Merkmal `OBJECTINFO` des Pronomens *it* unifiziert, woraufhin eine Anfrage an die Inferenzmaschine konstruiert wird, ob das besagte Objekt die Eigenschaft *huge* besitzt.

8.4.3 Beschränkungen

Diese Vorgehensweise bietet nur eine sehr einfache Form der Anaphernauflösung. So kommt es beispielsweise bei folgenden Sätzen zu Problemen: "*There is a blue block and there is a red block. They are huge.*" Eine solche Gruppierung ist momentan nicht von der nichtkompositionellen Semantik auflösbar. Ähnliche Probleme bereitet der Satz "*The blue ball or the red ball is large.*", da das Semf-System generell nicht mit Disjunktionen umgehen kann.

Ein sehr komplexes Zusammenspiel zwischen den Moduln nichtkompositionelle Semantik, Inferenzmaschine, Master und Generalisierung wäre notwendig, um den folgenden Satz verarbeiten zu können: "*There is a blue ball.*" In der momentanen Blocksweltkonfiguration existiert jedoch kein blauer Ball. Eine Rückfrage an den Schüler erscheint hier sinnvoll: "*There is a red ball and a yellow ball. Which one do you mean?*"

Wünschenswert wäre des weiteren die Analyse des Bezuges auf ein Objekt aus einer Menge: "*There are some blocks. The biggest one is red.*"

8.5 Quantorenbehandlung

Stellt man dem Benutzer eines Sprachlehrsystems die Aufgabe, eine auf dem Bildschirm abgebildete Welt – in unserem Falle die Blockswelt – zu beschreiben, so wird man zwangsläufig mit dem bekannten Problem von mehrdeutig interpretierbaren Quantoren konfrontiert. So ist beispielsweise der Satz "*All pyramids are lying on a cube*" auf zwei verschiedene Weisen interpretierbar, die hier in prädikatenlogischer Form dargestellt werden:

1. $\exists y(cube(y) \wedge \forall x(pyramid(x) \rightarrow lying\_on(x, y)))$
2. $\forall x(pyramid(x) \rightarrow \exists y(cube(y) \wedge lying\_on(x, y)))$

In der ersten Lesart existiert genau ein Würfel, auf dem sich alle Pyramiden befinden; in der zweiten Lesart gibt es für jede Pyramide genau einen Würfel, auf dem sie liegt. Es ist durchaus wünschenswert, solche Ausdrücke in einem Sprachlehrsystem verarbeiten zu können. Kann ein System verschiedene Lesarten unterscheiden, so ist man – mit einem geeigneten didaktischen Konzept – in der Lage, dem Schüler den Unterschied zwischen diesen Lesarten nahezulegen.

8.6 Ausblick

Die nichtkompositionelle Semantik ist eine sehr spezialisierte Komponente des Semf-Systems; sie leistet drei verschiedene Aufgaben: das Umschreiben des Formats der Parserausgabe in die Anfragesprache der Inferenzmaschine, das Auflösen einfacher anaphorischer Bezüge sowie eine sehr rudimentäre Quantorenskopus-Bestimmung. Da im Falle der beiden letztgenannten Punkte andere Moduln des Semf-Systems wesentlich zur Funktionalität beitragen (z.B. der Parser und das Lexikon), erscheint es hier sehr aufwendig, komplexere Verfahren – zum Beispiel eine Anaphernresolution nach dem Vorbild der DRT – zu integrieren.

Kapitel 9

Inferenzmaschine
Andreas Rother

9.1 Inferenzmaschine

Die Inferenzmaschine verbindet Master und Weltrepräsentation. Anfragen zum Zustand der Welt werden ausgewertet, die Ergebnisse an den Master zurückgeliefert. Dabei können alle vorhandenen Informationen zur gegenwärtigen Konfiguration der Welt und ihrer Objekte abgefragt werden:

- Typ
- Farbe
- Größe
- Orientierung
- Relationen zwischen Objekten

Relationen zwischen zwei oder mehreren Objekten werden unterschieden. Anfragen können geschachtelt werden, das heißt Zwischenergebnisse einer Anfrage werden zur weiteren Analyse benutzt. Der Umfang der Analysemöglichkeiten ergibt sich aus dem Informationsbedarf des Gesamtsystems. Prinzipiell besteht jedoch die Möglichkeit, über Anfragen hinaus Veränderungen an der Konfiguration einer Welt vorzunehmen. Die Generierung neuer Welten ist dabei genauso möglich wie das Einfügen neuer Entitäten in die bestehende Objekthierarchie. Alle Aspekte der vorhandenen Objekte können modifiziert werden. Ebenso die Anordnung der Objekte in betreffenden Welten.

Die Inferenzmaschine nutzt die Datenstrukturen, die von der Weltrepräsentation verwendet bzw. zur Verfügung gestellt werden. Um die Anfragen auswerten zu können ist daher eine Übersetzung der Anfragesprache in den Formalismus der Repräsentation zu leisten. Dabei handelt es sich um einen mehrstufigen Prozeß. Nach dem Parsen der Struktur werden die Anfragen in ihre Teilstrukturen zerlegt, die dann selbst entsprechend der ihnen zugrundeliegenden Hierarchie abgearbeitet werden. Zu diesem Zweck finden zwei getrennt arbeitende Parser Verwendung.

Das Format der Ergebnisse orientiert sich zweckmäßigerweise stark an der Form der Auswertung, sowie an den Bedürfnissen des Semf–Systems. Durch die strikte Modularisierung der Inferenzmaschine steht einer Anpassung an andere Systeme hier jedoch nichts im Weg.

9.1.1 Schnittstellen

Die Inferenzmaschine verfügt über Schnittstellen zum Master und zur Weltrepräsentation. Es besteht die Möglichkeit, auch von ausserhalb des Semf–Systems auf die Funktionalität der Inferenzmaschine zurückzugreifen.

Schnittstelle zum Master

Weltrepräsentation und Inferenzmaschine sind mit dem Anspruch entwickelt worden, nicht nur speziell innerhalb des Semf–System eingesetzt werden zu können. Darüber hinaus sollte ein einfaches Testen der Komponente in einer minimalen Testumgebung ermöglicht werden. Um diesem Anspruch zu genügen wurde die Schnittstelle zum Master als Textschnittstelle realisiert. Die

Kommunikation findet über einen UNIX–Socket statt, wodurch der Zugriff durch einfache Tools wie `telnet` ebeso möglich ist wie die Integration in ein anderes System.

Die eingehenden Anfragen müssen in die Datenstrukturen der Inferenzmaschine übersetzt werden. Durch Verwendung zweier Parser findet die Analyse der Eingabe schrittweise statt. Zunächst wird die Oberflächenstruktur der Anfrage geparsed und die Formel in Teilformeln zerlegt. Danach werden die entstandenen Anfragen ausgewertet und das Ergebnis jeder einzelnen, sowie der gesamten Anfrage konstruiert. Der zweite Parser dient also der Auswertung der Teilformeln. Dieser Mechanismus ist in der Literatur bekannt als *recursive descent* und wird zur Auswertung mathematischer Formeln verwendet.

Schnittstelle zur Weltrepräsentation

Die Inferenzmaschine arbeitet auf den Datenstrukturen der Weltrepräsentation. Zusätzlich werden einige neue Strukturen hinzugefügt, die zu Analysezwecken während der Inferenz benötigt werden. Die Schnittstelle stellen die Klassen `World` und `Inference` dar. Informationen über Relationen werden der Klasse `Relation` entnommen, Attributinformationen befinden sich in den von `Entity` abgeleiteten Klassen. Durch öffentliche Methoden wird der Zugriff der Infernezmaschine auf die Daten der o.g. Klassen gewährleistet.

9.1.2 Anfragesprache

Die folgenden Anfragen geben einen Überblick über die Möglichkeiten, die das gewählte Format bietet. Stark vereinfacht folgt einer Objektbeschreibung eine Relation und dann wieder eine Objektbeschreibung. *Pyramid* und *Floor* sind in diesem Beispiel Bezeichner für Entitäten, die prototypisch implementiert worden sind. Prinzipiell besteht die Möglichkeit, das System zur Laufzeit über die bereits zur Verfügung stehenden Klassen hinaus zu erweitern. In den vorliegenden Beispielanfragen wird jedoch vorausgesetzt, daß die bezeichneten Objekte implementiert sind:

Befinden sich mehr als die Hälfte aller Pyramiden auf dem Boden

```
? (Pyramid 1 %50 %100 e|on|Floor 2 1 1 e)
```

```
(4 8) [0] Pyramid 1 %50 %100 e
(1) [1] Floor 2 1 1 e
```

`Pyramid` 4 und 8 werden in Cluster 0 zusammengefaßt. `Floor` 1 gehört zu Cluster 1.

```
(4 [0] on 1 [1] 0)
(8 [0] on 1 [1] 1)
```

`Pyramid` 4 befindet sich nicht in der Relation `on` zu `Floor` 1. `Pyramid` 8 befindet sich in der Relation `on` zu `Floor` 1.

```
[0] <1 2> 2 1
[1] <1 1> 1 1
```

Der Range der Anfrage in Cluster 0 ist das Intervall von 1 bis 2. Zwei Elemente aus Cluster 0 sind vorhanden, eines davon erfüllt die geforderte Relation (`on`).

```
ok {8}
```

Das Resultat der Anfrage lautet `ok`, `Pyramid` 8 erfüllt die Relation `on` in Bezug auf `Floor` 1.

Im ersten Teil der Anfrage wird eine Entität der Klasse *Pyramid* spezifiziert, sie erhält das Tag 1. 50 bis 100 Prozent aller existierenden Pyramiden müssen dabei in der Relation *on* zu einer Entität der Klasse *Floor* mit dem Tag 2 stehen. Der Begriff *Range* bezeichnet das entsprechenede Intervall, das durch prozentuale oder absolute Zahlen spezifiziert wird. Die fünfte Stelle der Anfrage (hier: *e*) bedeutet, daß auf eine Entität Bezug genommen wird, nicht jedoch auf eine Gruppe von Entitäten[1]. Die Bedeutung der Begriffe *Cluster* und *Tag* wird in Zusammenhang mit dem Inferenzalgorithmus erläutert.

Im folgenden Beispiel wird die Anfrage um die Spezifikation verschiedener Attribute ergänzt. Durch diese Spezialisierung kann der Suchraum eingeschränkt werden:

Befindet sich genau ein Würfel mit den Attributen `color=yellow` *und* `size=small` *auf der Entität mit der* `id` *1*

```
? (Cube 3 1 1 e color=yellow,size=small|on|1)
```

```
(7) [0] Cube 3 1 1 e color=yellow,size=small
```

`Cube` 7 befindet sich in Cluster 0.

```
(7 [0] on 1 1)
```

`Cube` 7 befindet sich in der Relation `on` zu 1.

```
[0] <1 1> 1 1
```

Der Range der Anfrage in Cluster 0 ist das Intervall von 1 bis 1. Ein Element aus Cluster 0 ist vorhanden, es erfüllt die geforderte Relation `on`.

```
ok {7}
```

Das Resultat der Anfrage lautet `ok`, `Cube` 7 erfüllt die Relation `on` in Bezug auf 1.

Die Anzahl der bezeichneten Attribute ist nicht beschränkt. Man kann in Form einer Liste beliebige und beliebig viele Attribute angeben. Für die Attribute gilt also prinzipiell wie auch für die Entitätsklassen keine Einschränkung hinsichtlich der Bezeichner. Auch Attribute können zur Laufzeit durch Hinzufügen neuer Entitätsklassen ergänzt werden[2].

[1] Siehe Weltrepräsentation, räumliche Relationen zwischen Objekten und Gruppen.

[2] Die Erweiterung der Bedeutung bereits eingeführter Attribute oder deren Kombination wäre ebenfalls wünschenswert.

9.1.3 Algorithmus

Der Inferenzalgorithmus arbeitet in mehreren Phasen:

1. Die Formel wird in Teilformeln zerlegt.
2. Eine Liste mit Objekten, auf die die gegebene Beschreibung zutrifft, wird generiert.
3. Die Relationen werden überprüft.
4. Die Ranges werden überprüft.
5. Ein Zwischenergebnis wird generiert. Gegebenenfalls wird rekursiv bei 2. wieder aufgesetzt, bis alle Teilformeln abgearbeitet worden sind.

Die Zerlegung in Teilformeln wird innerhalb der Klasse `InferenceParser` geleistet. Die Methode `inference()` identifiziert Trennsymbole und leistet eine rekursive Analyse der Formel. Sobald eine Teilformel identifiziert worden ist, wird diese einer Entität der Klasse `Inference`. Methode `inferStatement()` zur weiteren Auswertung übergeben. Mit dem Zwischenergebnis wird dann weitergerechnet, bis alle Teilformeln ausgewertet worden sind. Daraufhin wird das Endergebnis der Inferenz zurückgeliefert.

Auch die Analyse der Teilformeln erfolgt schrittweise. Zunächst werden Objektlisten erstellt. Diese beinhalten Informationen über die Objekte, die aufgrund der Objektdefinition identifiziert werden konnten. Zur Identifikation dienen Objektklasse und die zugehörige Attributliste. Der Tag dient ausschließlich der eindeutigen Identifikation der Objekte innerhalb des Resultats, da es sich dabei wiederum um einen Text handelt, der geparsed werden muß. Der Range muß zu einem späteren Zeitpunkt berechnet werden, falls prozentuale Angaben enthalten waren.

Daraufhin werden die Relationen geprüft. Es gibt ein–, zwei– und dreistellige Relationen. Dabei wird die Prüfung auf Existenz eines Objekts als einstellige Relation aufgefaßt (`exist`). Alle weiteren Relationen bis auf `between` sind zweistellig. Es werden nun Listen erstellt, die Informationen über den Grad des Zutreffens einer Relation auf zwei Objekte oder Objektgruppen beinhalten. Das Ergebnis dieser Prüfungen liegt dabei im Intervall von 0 bis 1[3].

An dieser Stelle wird auch die Negation behandelt. Das Ergebnis der Relationsprüfung wird dann ggf. von 1 subtrahiert[4].

Im nächsten Schritt werden die Ranges überprüft. Dazu werden die bis dahin gesammelten Informationen benötigt. Um aus Prozentangaben absolute Werte zu berechnen muß zuvor die Anzahl der für die entsprechende Relation relevanten Objekte ermittelt worden sein. Das Intervall von 0 bis 100 Prozent kann jetzt also auf das Intervall der tatsächlich vorhandenen Objekte abgebildet werden[5]. Um die Anfrage bewerten zu können müssen jetzt die Anzahlen aus der Anfrage mit den ermittelten verglichen werden. Eine Anfrage wird dann positiv bewertet, wenn die Anzahl

[3]Siehe Weltrepräsentation, räumliche Relationen zwischen Objekten und Gruppen.

[4]Eine ausführlichere Behandlung der Negation wäre wünschenswert, da sich auf diese Art ausschließlich externe Negation behandeln läßt.

[5]In wieferen diese Methode adäquate Ergebnisse in Hinblick auf sehr kleine Testwelten liefert soll hier nicht diskutiert werden.

der in der geforderten Relation befindlichen Objekte und Objektgruppen innerhalb des Intervalls der Anfrage liegen. Dieselbe Bewertung wird auch bei Zwischenergebnissen vorgenommen. Sollte allerdings bei einem Zwischenergebnis das o.g. Kriterium nicht erreicht werden, so wird mit den Objekten und Objektgruppen weitergerechnet, die die geforderte Relation erfüllen:

Sind 50 Prozent aller Bälle größer als der Block auf dem Floor

```
? (Ball 4 %50 %50 e|bigger|
    (Block 5 1 1 e|on|Floor 6 1 1 e))

(2 3 7 9) [0] Block 5 1 1 e
(1) [1] Floor 6 1 1 e
(2 [0] on 1 [1] 1)
(3 [0] on 1 [1] 1)
(7 [0] on 1 [1] 1)
(9 [0] on 1 [1] 1)
[0] <1 1> 4 4
[1] <1 1> 1 1
fail {2+3+7+9}
(5 6) [0] Ball 4 %50 %50 e
(5 [0] bigger 2 0)
(5 [0] bigger 3 0)
(5 [0] bigger 7 0)
(5 [0] bigger 9 0)
(6 [0] bigger 2 0)
(6 [0] bigger 3 0)
(6 [0] bigger 7 1)
(6 [0] bigger 9 1)
[0] <1 1> 2 1
ok {6}
```

In dieser Anfrage wird als Zwischenergebis `fail {2+3+7+9}` geliefert. Dennoch lautet das Endergebis `ok` {6}, da nach dem negativen Zwischenergebnis mit den Blöcken 2, 3, 7 und 9 weitergerechnet wurde. Das Zwischenergebnis resultiert aus dem Range, der auf genau einen Block verweist, obwohl vier Blöcke identifiziert werden konnten, die auf dem Floor liegen. Einer von zwei vorhandenen Bällen (50 Prozent) ist größer als die Blöcke 7 und 9, das Endergebis ist daher also positiv zu bewerten[6].

In eckigen Klammern befinden sich Clusternummern, die während der Analyse erzeugt werden, um einzelne Objekte zusammenzufassen. Cluster können sowohl einzelne Objekte (bzw. deren `ids`), als auch Objetgruppen enthalten. Durch diese Indizes ist es möglich, eine Bewertung der Ranges vorzunehmen, da die Anzahl der vorhandenen Objekte je Cluster bekannt ist.

[6] Es ist fragwürdig, ob dieses Verfahren adäquat ist.

9.2 Sprachbeschreibung

Die Syntax der Anfragesprache läßt sich formal folgendermaßen beschreiben:

```
<Anfrage> = (<TeilAnfrage>) ‖
     ([<Gruppe>+]<Anfrage>[+<Gruppe>]) ‖
     (<Anfrage>|<Relation>|<Anfrage>)

<TeilAnfrage> = <Negation> <Gruppe>|<Relation>
     [|<Gruppe>[|<Relation>|<Gruppe>] <AnfrageMarker>]

<Relation> = on‖beneath‖front‖behind‖above‖bigger‖higher‖smaller‖
     <Relation>+<Relation>

<Gruppe> = <Objekt>‖<Gruppe>+<Objekt>

<Entität> = Entity‖Block‖Cube‖Pyramid‖Ball‖Floor

<Objekt> = <Entität> <Tag> <Quantor>‖<Id>

<Id> = id | id ∈ {1, 2, 3, ...}

<Tag> = tag | tag ∈ {0, 1, 2, ...}

<Quantor> = <Absolut>‖<Relativ> <Absolut>‖<Relativ> <QuantorMarker>

<Absolut> = n | n ∈ {1, 2, 3, ...}

<Relativ> = %p | p ∈ {0, ..., 100}

<Negation> = not‖∅

<QuantorMarker> = e‖g

<AnfrageMarker> = E‖G‖∅
```

Legende:

‖	Exklusives *oder*
+, |	Trenner
< >	Inhalt ist ein Bezeichner
[ ]	Inhalt kann entfallen oder genau einmal vorkommen

Kapitel 10

Weltrepräsentation

Christoph Peylo

10.1 Weltrepräsentation

Aufgabe der Weltrepräsentationskomponente ist es, alle erforderlichen Objekte und Operationen zur Verfügung zu stellen, um die Modellierung eines konsistenten Weltausschnitts zu ermöglichen. Ein Weltausschnitt definiert sich in diesem Zusammenhang als eine mögliche Welt sowie alle Festlegungen und Gesetze, die dort herrschen.

Innerhalb des Systems können beliebige Welten erzeugt werden, für die sich jeweils charakteristische Gesetze einführen lassen. Die gegenwärtige Weltmodellierung ist auf die Erfordernisse des momentanen Systems beschränkt, ein Ausbau ist allerdings jederzeit möglich, ohne die Konsistenz des Bisherigen zu gefährden.

Die Weltrepräsentation stellt Raum, Objekte und alle Operationen zur Verfügung, die zum räumlichen Schließen benötigt werden. Die Objekte werden dabei piktoriell repräsentiert, auf diese Weise sind alle räumlichen Relationen schon im Aufbau der Welt enthalten.

10.2 Ontologie

Wenn im Folgenden von „Ontologie“ die Rede sein wird, ist dabei der Aufbau und die Struktur einer möglichen Welt gemeint. D.h. was das Wesen der in der Welt vorhandenen Entitäten ist und wie sich diese zueinander verhalten.

Die hier verwendete Ontologie ist eine Spielart des Konzeptualismus, d.h. nur konkrete Objekte existieren im eigentlichen Sinne, Allgemeinbegriffe und Abstraktionen über Entitäten sind nicht wirklich real. Sie wird im wesentlichen durch die Klassen `Attribute, Coordinate, World, Entity` und `Group` bestimmt. Durch `Coordinate` läßt sich für jedes Objekt ein Bezugspunkt im Raum festlegen, und mit `Attribute` können einem Objekt Attribute zugeordnet werden. Eine Welt (`World`) ist in erster Linie ein Vektor, der all das enthält, was innerhalb der genannten Welt der Fall ist. Hierzu zählen natürlich die Einzeldinge und Gruppierungen von Einzeldingen, es können aber auch spezielle Relationen festgelegt werden, dir nur innerhalb von einer spezifischen Welt gelten. Im gegenwärtigen System ist bislang nur der Raum modelliert, zeitliche und dynamische Aspekte wurden noch nicht behandelt. Die räumliche Darstellung basiert auf einem dreidimensionalen Koordinatensystem mit x-, y- und z-Achse. Ein Würfel mit der Kantenlänge 1 ist die kleinste mögliche Einheit innerhalb des Systems. Aus diesen kleinsten Einheiten, sogenannten Einheitswürfeln, ergibt sich die räumliche Ausdehnung eines Objekts. Die äußeren Grenzen eines Objekts bestimmen, welche Einheitswürfel innerhalb und welche außerhalb seiner Grenzen liegen. Die Einheitswürfel eines Objekts werden in einem Vektor innerhalb des Objekts verwaltet.

Das Wesen eines Objekts wird in der Klasse `Entity` festgelegt. Entity enthält `Coordinate` und `Attribute` als Voraussetzungen, daß jedes Einzelding räumliche Ausdehnung besitzen und gewisse Attribute (wie Größe, Farbe, etc. ...) haben kann. Von dieser Klasse sind alle Klassen abgeleitet, von denen es innerhalb der Welt konkrete Instanzen gibt (wie `Ball, Pyramid, Block` und `Cube`). `Entity` stellt somit einen Allgemeinbegriff dar, der über alle konkreten Objekte abstrahiert und dem keine eigentliche eigene Existenz zukommt

Neben Einzeldingen gibt es auch Zusammenfassungen von Einzeldingen zu größeren Entitäten, zu einer `Group`. Eine `Group` läßt sich ebenfalls als Abstraktion über Einzeldingen verstehen, d.h. sie existert selbst nur durch ihre Komponenten. Aus diesem Grunde wird, wenn eine Gruppe gelöscht wird, lediglich der Verweis auf die Gruppe entfernt nicht aber alle Objekte, die in der Gruppe zusammengefaßt sind. Bei dem momentanen Stand des Systems entspricht der Begriff der Gruppe dem Mengenbegriff, daher sind auch Mengenoperationen als Gruppenmethoden implementiert.

10.3 Das Einzelne und die Gruppe

10.3.1 Das Einzelne – Entity

Nachdem ein Überblick über die Ontologie des Systems gegeben wurde, soll im Folgenden noch einmal genauer auf die Modellierung einzelner Entitäten und die Zusammenfassung mehrerer Entitäten zu Gruppen eingegangen werden.

Die Implementation der Welt erfolgt objektorientiert. Bei dieser Sichtweise werden die einzelnen Objekte als eigenständige Entitäten behandelt, die sich durch ihren Zustand und ihre Aktionen bzw. Aktionsmöglichkeiten unterscheiden. Der Zustand eines Objekts wird durch seine Variablen bestimmt, die möglichen Aktivitäten durch Methoden. Eine Methode entspricht einem Funktionsaufruf, bei dem das implizite erste Argument der Aufzurufende darstellt. Eine Klasse stellt einen Allgemeinbegriff dar, also etwas Abstraktes, während ein Objekt eine konkrete Instanz einer Klasse ist.

Die Klasse `Entity` verfügt über mehrere Variablen, in denen sich Attribute, d.h. Angaben über Farbe, Größe, Charakteristiken und Zustände eines Objekts festlegen lassen. Ferner besteht die Möglichkeit, für jede vom System unterstützte Relation objektspezifische Heuristiken zu formulieren. In jeder Entität ist weiterhin ein Bezugspunkt angelegt, der diversen Berechnungen zugrunde gelegt wird, sowie ein Vektor, in dem die Einheitswürfel, die die räumliche Ausdehnung eines Objekts realisieren, abgelegt sind.

Eine Entität besitzt Methoden, die die Verwaltung und Manipulation ihrer Variablen durch andere Objekte ermöglichen. Auf diese Methoden muß an dieser Stelle aber nicht ausführlicher eingegangen werden. Interessanter sind hier die Methoden, die Auskunft über die entsprechende Entität geben oder Operationen auf ihr ermöglichen. So kann eine Entität ihre Ähnlichkeit zu einer anderen Entität berechnen, Auskunft über den von ihr beanspruchten Raum geben, oder ihre euklidische Distanz zu einer anderen Entität berechnen. Um ein Objekt in eine Welt einfügen oder innerhalb einer Welt bewegen zu können, sind dementsprechende Methoden definier. Da vorhandener Raum nicht beliebig oft vergeben werden kann, ist eine Verwaltung des Raumes notwendig. Hier geschieht diese Verwaltung durch die einzelnen Objekte.

Wie bereits angesprochen, ist `Entity` die Oberklasse aller Objekte, die tatsächlich in der Welt existieren. Die Unterklassen, wie `Ball`, `Cube`, `Pyramid` und `Block` erweitern die von `Entity` zur Verfügung gestellten Methoden und Variablen je nach ihren individuellen Erfordernissen. Die konkreten Einzeldinge entstehen durch Instanziierungen der jeweiligen Unterklassen. Einzelne Instanzen einer Klasse unterscheiden sich durch unterschiedliche Attributwerte, in der Regel Größe und Farbe, zumindest aber durch die räumliche Position.

10.3.2 Die Gruppe – Group

Wie bereits erwähnt, wird durch `Group` die Zusammenfassung einzelner Entitäten zu Objekten anderer Qualität realisiert. Wenn beispielsweise `Wolf` eine von `Entity` abgeleitete Klasse ist, dann lassen sich mit `Group` Instanzen von `Wolf` zu einem `Rudel` zusammenfassen. Als Instanzvariablen enthält `Group` einen Vektor, der die Mitglieder der Gruppe enthält. Da eine `Group` etwas qualitativ anderes als die Summe ihrer Mitglieder ist, kann eine Gruppe eigene Attribute besitzen, die vom Typ her in etwa denen von `Entity` entsprechen.

Neben den Methoden, die die Variablen verwalten, besitzt `Group` Methoden, durch die sich Gruppen aufbauen und verwalten lassen. Eine Gruppe wird durch bestimmte Gemeinsamkeiten konstituiert. So lassen sich Gruppen aufgrund gemeinsamer Attribute, die konjunktiv oder disjunktiv verknüpft sein können, oder gemeinsamer Klassenzugehörigkeit bilden. Wenn die Objekte bestimmt sind, die zu der Gruppe zugehören sollen, werden die Objekte durch die konvexe Hülle auch geometrisch verbunden, um räumliches Schließen zwischen Objekten und Gruppen zu ermöglichen. Durch weitere Methoden kann von `Group` bestimmt werden, ob sich ein beliebiges Objekt innerhalb seiner konvexen Hülle befindet. Ferner sind Mengenoperationen wie die Bildung von Vereinigungs- oder Schnittmenge zweier Gruppen realisiert.

10.4 Die räumlichen Relationen

Für das gegenwärtige System sind folgende Relationen als Relationen zwischen einzelnen Objekten, zwischen einzelnen Objekten und Gruppen sowie zwischen Gruppen realisiert:

touch: Um herauszufinden, ob sich zwei Entitäten seitlich berühren, wird, eine Entität um jeweils einen Einheitswürfel nach rechts und nach vorne sowie nach links und nach hinten zu verschoben. Führt eine der Verschiebeoperationen zu einer Kollision beider Objekte, so müssen sie sich davor berührt haben.

Als Relation zwischen einem Objekt o und einer Gruppe G mit $g_1, g_2, g_3 \dots g_n$ wird berechnet, für wieviele Objekte g_i, mit $1 \leq i \leq n$, die Relation $\texttt{touch}(o, g_i)$ gilt. Das Ergebnis wird durch eine reele Zahl ausgedrückt, die das Verhältnis zwischen den Objekten g_i angibt, bei denen besagte Relation gilt, bezüglich aller Objekte aus G.

Die Relation ist zwischen zwei Gruppen G_1 und G_2 erfüllt, wenn jedes Mitglied von G_1 ein Objekt von G_2 berührt. Eine reele Zahl, die als Resultat der Relation zurückgegeben wird, spezifiziert, in welchem Maße die Relation erfüllt ist.

on: Zwei Objekte befinden sich aufeinander, wenn die oberste Lage von Einheitswürfeln des einen Objekts die unterste Lage von Einheitswürfeln des anderen Objekts berührt.

Analog zum Vorgehen bei der Relation `touch` wird errechnet, wieviele Mitglieder der Gruppe G in diesem Verhältnis zu dem einzelnen Objekt o stehen.

Die Relation ist zwischen zwei Gruppen G_1 und G_2 erfüllt, wenn sich jedes Mitglied von G_1 auf einem Objekt von G_2 befindet. Hierbei spielt es keine Rolle, ob sich die Objekte aus G_1 auf dem selben oder jeweils einem anderen Mitglied aus G_2 befinden. Eine reele Zahl, die als Resultat der Relation zurückgegeben wird, drückt aus, in welchem Maße die Relation erfüllt ist.

beneath: Für alle $a, b \in \{\texttt{Group}, \texttt{Entity}\}$ gilt: `on(a,b)` $\leftrightarrow$ `beneath(b,a)`.

nextTo: In gewisser Weise ist dies die schwierigste Relation, da ihr keine allgemeingültigen Kriterien zugrunde liegen. Ob zwei Objekte als nebeneinander befindlich empfunden werden, hängt von verschiedenen Faktoren ab:

- Von der Entfernung der Objekte zueinander. Hierbei spielt weniger die absolute Entfernung der Objekte eine Rolle, als ihre Entfernung bezüglich der durchschnittlichen Entfernung aller Entitäten in dem entsprechenden Weltausschnitt zueinander. So können beispielsweise auf freiem Feld zwei Bäume als in der Nähe von einander stehend angesehen werden, die bei gleicher Entfernung in einem Wald nicht mehr als unmittelbar benachbart empfunden werden würden.
- Ferner spielt die Größe der Objekte eine Rolle. Je kleiner ein Objekt ist, desto näher muß es an einem anderen Objekt liegen, damit diese Relation gilt.
- Weiterhin gibt es klassenspezifische Unterschiede. Objekte können bei Betrachtern Assoziationen wecken, die dazu führen, daß die „Einflußbereiche" von äußerlich vergleichbaren Objekten ganz unterschiedlich bemessen werden.
- Schließlich gibt es auf Seiten des Beobachters ganz erhebliche individuelle Unterschiede.

Wenn auch die genauen Mechanismen, die zu den bestimmten Zuordnungen führen, noch unbekannt sind, wurden bereits in den 30er Jahren dieses Jahrhunderts in der Psychologie Regeln (*Gestaltgesetze*) formuliert, die sich mit diesem Phänomen befassen.[1] Um die Relation möglichst adäquat modellieren zu können, wurde hier unter Berücksichtigung der Gestaltgesetze folgendermaßen verfahren:

- Zwei einzelne Objekte befinden sich in der Relation `nextTo(a,b)`, wenn ihre Entfernung multipliziert mit einem klassenspezifischen und einem größenspezifischen heuristischen Wert zueinander geringer ist als die mittlere Euklidische Distanz aller Objekte zueinander.
- Eine einzelne Entität o ist in der Nähe einer Gruppe g, wenn der euklidische Abstand des Bezugspunkt von o nicht größer als der mittlere euklidische Abstand (multipliziert mit einer größen- und klassenspezifischen Konstante) aller Objekte zueinander ist.
- Zwei Gruppen G_1, G_2 befinden sich in der Relation `nextTo(`G_1`,`G_2`)`, wenn der euklidische Abstand der Mittelpunkte von G_1, G_2 nicht größer als die mittlere euklidische Distanz aller Objekte multipliziert mit einer klassen- und größenspezifischen Konstante ist.

[1] siehe hierzu Günther Kebeck: „Wahrnehmung, Theorien, Methoden und Forschungsergebnisse der Wahrnehmungspsychologie". Juventa, Weinheim und München 1994, S. 41ff.

front: Zwei einzelne Objekte a, b stehen zueinander in der Relation `front(a,b)`, wenn der z-Wert des Bezugspunkts von a kleiner als der von b ist und die Relation `nextTo(a,b)` erfüllt ist.

Ein Objekt o kann mit einer Gruppe G mit g_1, g_2, $g_3 \dots g_n$ in der Relation $\texttt{front}(o, G)$ stehen, wenn die Relation $\texttt{nextTo}(o, G)$ erfüllt ist. Der Grad, in dem die Relation zwischen o und G zutrifft, wird durch eine reele Zahl ausgedrüßckt, die angibt für wieviele Objekte g_i, mit $1 \leq i \leq n$, die Relation $\texttt{front}(o, g_i)$ gilt.

Eine Gruppe G_1 befindet sich vor einer Gruppe G_2, wenn:

- sich die Gruppen in der Relation $\texttt{nextTo}(G_1, G_2)$ befinden,
- sich der Bezugspunkt von G_1 vor dem Bezugspunkt von G_2 befindet,
- die Gruppen voneinander separierbar sind. D.h. es dürfen sich nicht beliebig viel Mitglieder der einen Gruppe innerhalb der konvexen Hülle der anderen Gruppe befinden.

behind: Für alle $a, b \in \{\texttt{Group}, \texttt{Entity}\}$ gilt: $\texttt{behind(a,b)} \leftrightarrow \texttt{front(b,a)}$.

side: Die Relation gilt für zwei Entitäten a, b, wenn der x-Wert des Bezugspunkts von a kleiner oder größer als der von b ist und beide in der Relation `nextTo(a,b)` zueinander stehen.

Analog zu den anderen Relationen kann ein Objekt o mit einer Gruppe G mit $g_1, g_2, g_3 \dots g_n$ in der Relation $\texttt{side}(o, G)$ stehen, wenn zunächst die Relation $\texttt{nextTo}(o, G)$ erfüllt ist. Der Grad, in dem die Relation $\texttt{side}(o, G)$ zutrifft, wird durch eine reele Zahl ausgedrückt, die angibt für wieviele Objekte g_i, mit $1 \leq i \leq n$, die besagte Relation gilt.

Eine Gruppe G_1 befindet sich seitlich einer Gruppe G_2, wenn:

- sich die Gruppen in der Relation $\texttt{nextTo}(G_1, G_2)$ befinden,
- sich der Bezugspunkt von G_1 seitlich von dem Bezugspunkt von G_2 befindet,
- die Gruppen voneinander separierbar sind.

between: Wenn drei Objekte a, b, c in den Relationen `behind(a,b)` und `behind(c,a)` bzw. `front(b,a)` und `front(a,c)` oder `side(a,b)` und `side(a,c)` zueinander stehen, wobei der x-Wert von a größer als der von b und kleiner als der von c sein muß, besteht die Relation `between(a,b,c)`.

Ein Objekt kann sich zwischen einer Gruppe befinden, wenn es sich in der konvexen Hülle einer Gruppe befindet. Es befindet sich zwischen zwei Gruppen, wenn die Relation `between` zwischen den Bezugspunkten der Gruppe und dem Objekt gilt. Um zu untersuchen, ob sich eine Gruppe G zwischen zwei Entitäten o_1 und o_2 befindet, wird für jedes Mitglied der Gruppe getestet, ob die Relation $\texttt{between}(o_1, g_i, o_2)$ erfüllt ist (mit g_1, g_2, $g_3 \dots g_n$ und $1 \leq i \leq n$). Der Grad, in dem die Relation zutrifft wird wieder durch eine reele Zahl angegeben.

Eine Gruppe G_1 befindet sich zwischen einer Gruppe G_2, wenn sich die meisten Glieder von G_1 innerhalb der konvexen Hülle von G_2 befinden. Eine Gruppe befindet sich zwischen zwei Gruppen, wenn die Gruppen voneinander separierbar sind und für ihre jeweiligen Bezugspunkte die die Relation `between` erfüllt ist.

10.5 Schnittstellen

Die Anbindung der Weltrepräsentation an das übrige System geschieht durch die Inferenzmaschine. Durch den Austausch von Nachrichten ist es der Inferenzmaschine möglich, auf den Datenstrukturen der Weltrepräsentation zu arbeiten.

Kapitel 11

Fehlerbehandlung

11.1 Allgemeines *Henrik Dittmann*

Ein wesentliches Defizit der kommerziell verfügbaren Sprachlehrsysteme stellen deren oftmals unzureichende Möglichkeiten zur Reaktion auf Fehler dar, die der Lerner in den Übungen macht. Während auf der einen Seite eine rein antizipationsbasierte Fehlerbehandlung nicht der Kreativität der Schülerin bei Bildung und Gebrauch von inkorrekten Formen, Wörtern und Satzstrukturen Rechnung tragen kann, scheitern fehlersensitive Parsingmethoden gerade an der Vielzahl und Komplexität der Fehlermöglichkeiten. So lassen sich für oberflächlich gleich aussehende Fehler sehr unterschiedliche Erklärungen angeben.[1] Durch diese alternativen Erklärungen in Verbindung mit der Möglichkeit des Auftretens mehrerer Fehler entstehen sehr große Suchräume, die einerseits die Geschwindigkeit der Syntaxanalyse dramatisch herabsetzen, andererseits durch die Vielzahl der Hypothesen eine Fehlererklärung nahezu unmöglich machen.

Es bietet sich daher an, im Fehleranalyseprozeß die rein syntaktische Verarbeitung der Lernerinneneingaben durch Wissen über den Kontext zu erweitern, in dem diese Eingaben gemacht werden. Informationen über das Curriculum, die Schwächen des einzelnen Schülers und die Ziele und Inhalte der aktuellen Übung erlauben es, Heuristiken aufzustellen, anhand derer die in der syntaktischen Bewertung erstellten Hypothesen bewertet bzw. bereits während der Analyse verworfen werden können.

Insbesondere eine semantische Verarbeitung der Eingaben ist vonnöten, da rein syntaxbasierte Methoden der Sprachanalyse nicht erkennen können, ob eine Äußerung im gegebenen Übungskontext überhaupt sinnvoll ist.

Ziel des *Semf*-Projektes war es[2], exemplarisch für einen bestimmten Übungstyp (Lernen räumlicher Präpositionen) semantische Informationen für die Fehleranalyse heranzuziehen. Während der syntaktischen Analyse des Eingabesatzes wird eine semantische Repräsentation erstellt, im Fehlerfall werden in Abhängigkeit von den jeweils durchzuführenden Fehlerkorrekturen eine oder mehrere Hypothesen über die von der LernerIn *intendierte Bedeutung* generiert (siehe Abschnitte 6.2, 11.2). Diese können anschließend an der systeminternen Repräsentation des der Übung zugrundeliegenden Weltausschnittes überprüft werden. Falls die erschlossene Bedeutung die Übungswelt nicht korrekt beschreibt, kann meist ein Abstand angegeben werden, der sich aus den Änderungen ergibt, die in der semantischen Repräsentation der Benutzereingabe vorgenommen werden müssen, um eine in der Welt gültige Aussage herzustellen (dazu mehr in Abschnitt 11.3).

Für die Rückmeldung an die Benutzerin wird diejenige Hypothese ausgewählt, die als die wahrscheinlich adäquate erscheint. Als Maß dafür werden folgende Kriterien berücksichtigt:

- Wahrscheinlichkeit und Schwere der syntaktisch notwendigen Fehlerkorrekturen
- Grad der Abweichung von der zu beschreibenden Weltdarstellung

Die Integration der von der syntaktischen und semantischen Fehlerbehandlung gelieferten Informationen sowie die Rückmeldung an den Lerner sind bisher nur rudimentär, zu Demonstra-

[1] Auf die Probleme der Diagnose und Klassifizierung von Lernerfehlern geht der Artikel *Fehler – Transfer – Interferenz* (Anhang 1) detailliert ein.

[2] vgl. Bongardt et al., *Projektantrag Semf.* Universität Osnabrück, 1996. S. 5f.

tionszwecken, implementiert. Die benötigte Funktionalität soll in Abschnitt 11.4 kurz skizziert werden.

11.2 Syntaktische Fehlerbehandlung *Michael Huber*

11.2.1 Einleitung

Dies Kapitel erläutert die vom Parser mit Hilfe von Grammatik und Fehlergrammatik durchgeführte Fehleranalyse und -diagnose.

Zunächst wird in einer kurzen Einleitung die Funktionalität der Parser-Fehlerbehandlungskomponente dargestellt, danach auf alle in Semf behandelten Fehlerklassen eingegangen.

Anschließend soll die Aufteilung der Fehlerklassen auf die beiden Module Grammatik und Fehlergrammatik dargestellt und motiviert werden, dabei werden deren jeweilige Rollen bei der Fehlererkennung erklärt und die Unterschiede und Gemeinsamkeiten beider Module deutlich gemacht. Wir erläutern die wichtigsten technischen Aspekte der syntaktischen Fehlerbehandlung: den Aufbau des `ERROR`-Features, dazu die bei der Fehlerbehandlung benutzten Funktionen. Schließlich soll die Arbeitsweise anhand von Beispielen verdeutlicht werden.

Allgemeines zur „syntaktischen" Fehlerbehandlung in Semf

Mit *syntaktisch* bezeichnen wir die während des Parsings stattfindende Fehleranalyse, ungeachtet dessen, daß der Parser Fehlerbeschreibungen sowohl für syntaktische Fehler als auch für andere Fehlerklassen erzeugt.

In geringem Maße ist die syntaktische Fehlerbehandlung im Lexikon verankert (siehe Kapitel 7); im wesentlichen aber basiert die Fehleranalyse und -diagnose unseres Systems auf Fehlerregeln in der Fehlergrammatik, darauf wird in diesem Kapitel fokussiert.

Beim Parsen werden durch Unifikation der Merkmalsstrukturen nicht nur Syntaxbaum und kompositionell-semantische Satzstruktur aufgebaut, sondern auch Fehlerdiagnose-Strukturen erzeugt. Weitgehend parallel zum Syntaxbaum, weisen auch diese zunächst eine Baumstruktur auf.

Eines der wesentlichen Probleme bei der Verwendung einer Fehlergrammatik ist die Kontrolle der Interaktion zwischen Regeln und Fehlerregeln: Durch einen Mechanismus zur Gruppierung von Regeln erreichen wir, daß bei Anwendbarkeit einer gewissen Regel dazugehörige Fehlerregeln blockiert werden, die allgemeiner als die erste Regel sind.

Eine Besonderheit unseres Systems ist die Tatsache, daß mehrere Fehler in einem Satz gleichzeitig behandelt werden können. Hinzu kommt die Fähigkeit, alternative Hypothesen über Fehler aufstellen zu können: wenn für eine Fehlerstelle im Eingabesatz mehrere Fehlerbeschreibungen möglich sind, werden automatisch mehrere Parses mit den jeweiligen alternativen Fehlerstrukturen erzeugt.

Die während des Parsings generierten Fehlerbeschreibungen sind zunächst auf die möglichst detaillierte Beschreibung der Fehler ausgerichtet. Sie werden nachträglich durch die Fehlerverbalisierung-Komponente aufbereitet, so daß Englischlernende etwas damit anfangen können. Alternativ wäre auch eine weitergehende Tutorkomponente denkbar, die noch andere Formen der Rückmeldung erlaubt.

Anmerkung

Die von unserem System geleistete Fehlerdiagnose ist unvollständig. Sowohl hinsichtlich der Abgrenzung der verschiedenen Fehlerklassen gegeneinander als auch der Auswahl der relevanten Fehlerphänomene bestehen schwerwiegende Probleme. Letzlich hoffen wir aber, mit unserem System dem noch jungen Forschungsgebiet der computerbasierten Fehlerdiagnose einige Anstöße geliefert zu haben.

11.2.2 Fehlerklassen in Semf

In Semf werden syntaktische, morphologische, lexikalische, orthographische und in einem gegebenen Kontext semantische Fehler behandelt. Die jeweiligen Instanzen der angegebenen Fehlerklassen repräsentieren – aus unserer Sicht – prototypische Fehler bezüglich des ausgewählten Sprachfragments, insbesondere für deutsche Muttersprachler, da auch spezifische Interferenzphänomene zwischen Deutsch und Englisch behandelt werden. Interferenzfehler können auf syntaktischer, orthographischer, morphologischer und lexikalischer Ebene auftreten; sie bilden also keine separate Fehlerklasse sondern liefern Hinweise auf Fehlerursachen. In sofern ist das Inferenzphänomen gerade für die Rückmeldung von Bedeutung und deshalb Bestandteil der Fehlerbeschreibung.

Anmerkungen bezüglich der im System eingeführten Restriktionen

Sicherlich ist es uns nicht in jedem Fall gelungen, die relevantesten Fehler exemplarisch zu behandeln, da jedoch hinsichtlich der Übungssituation eine Reihe von Beschränkungen vorgesehen sind, glauben wir, einen großen Teil der auftretenden Fehlerklassen abgedeckt zu haben. Wir möchten folgende Punkte in Erinnerung gerufen:

- die von uns generierte *blocks world* ist eine einfache, überschaubare Welt
- das der Welt zugrundeliegende Sprachfragment ist ebenfalls klein, insbesondere schlägt sich das im Lexikon nieder
- es werden nur einfache Aussagesätze als Eingaben zugelassen
- im Bereich der Verben gibt es nur die Präsensform
- in unserer Welt werden keine Aktionen ausgeführt, es fehlen also zeitliche Aspekte

Lexikalische Fehler

Hinsichtlich unserer Welt von Objekten und ihren Relationen werden in unserem System lexikalisch-präpositionale Fehler behandelt, die sich auf Domänenverletzung (*before* statt *in front of*) oder bestimmte idiomatisch bedingte Fügungen (**near to* statt *near*) beziehen, sowie Verwechslungen von Objekten (z.B. *block* mit *ball*) und deren Eigenschaften (z.B. eine falsche Farbe angeben).

Lexikalische Fehler sind insofern problematisch, als sie sich oft mit anderen Fehlerarten überlappen können, insbesondere mit semantischen und morphologischen Fehlern. Viele der behandelten lexikalischen Fehler sind ebenfalls potentielle semantische Fehler. Es wird jedoch davon ausgegangen, daß lexikalische Fehler in der Regel wahrscheinlicher sind als die dazu äquivalenten semantischen Fehler. Die Gründe dafür werden im nächsten Abschnitt angegeben.

Semantische Fehler

Verwechslungen von Objekten, deren Relationen und Eigenschaften können sowohl als lexikalische wie auch als semantische Fehler erklärt werden. Da unser Weltfragment sehr überschaubar ist, wird davon ausgegangen, daß es sich in vielen solchen Fällen eher um lexikalische als um semantische Fehler (also Verständnisprobleme) handelt. Nichtsdestotrotz gibt es auch rein semantische Fehler, z.B. innerhalb von Sätzen wie *Blue blocks are red.* Diese Fehlerklasse wird jedoch nicht durch den Parser, sondern von der Generalisierungskomponente behandelt. Semantische Fehler werden daher im Kapitel 11.3 ausführlicher diskutiert.

Syntaktische Fehler

Syntaktische Fehler werden in Kongruenz-, Rektions-, Wortstellungs-, Satzkonstruktions- und Subkategorisierungsfehler unterteilt. Aus jeder dieser Subklassen haben wir versucht, spezifische und relevante Fehlerinstanzen auszuwählen und sie durch geeignete Methoden zu analysieren.

Kongruenzfehler: In dieser Fehlerklasse (*agreement*) behandeln wir nur Inkongruenzen im Numerus zwischen Artikel und zugehörigem Nomen bzw. Nomen und dazugehörigem Verb, wobei jeweils eine der nicht kongruenten Konstituenten fehlerhaft sein kann.

Rektionsfehler: Innerhalb dieser Fehlerklasse (*government*) werden insbesondere Kasus-Verletzungen (z.B. *Them are big.*) behandelt. Es gibt ebenfalls Numerus-Verletzungen im Falle von Präpositionen, die einen bestimmten Numerus fordern (*between* fordert den Plural).

Wortstellungsfehler: In Semf werden hauptsächlich einfache Wortstellungsfehler behandelt, d.h. Vertauschungen von aufeinanderfolgenden Konstituenten (z.B. Adjektiv mit Nomen oder Nominalphrase mit Verbalphrase). Komplexere Wortstellungsfehler, Vertauschungen von nicht-adjazenten Konstituenten, haben wir nur exemplarisch anhand von Interferenzphänomenen untersucht, die solche Fehler hervorrufen (**the on the floor lying block* ← *der auf dem Boden liegende Block*).

Satzkonstruktionsfehler: Satzkonstruktionsfehler entstehen durch fehlerhaftes Hinzufügen oder Weglassen von Konstituenten. Auch diese Fehler können sehr komplex werden und sind schwierig zu behandeln. Wir haben uns hier wiederum auf einige Instanzen dieser Fehlerklasse konzentriert, die uns im Hinblick auf das Sprachfragment als sinnvoll erschienen. Beispiele dafür sind überflüssige Konjunktionen und Hilfsverben, oder fehlende Artikel.

Gerade Satzkonstruktionsfehler können sich häufig mit Kombinationen anderer Fehlertypen überlappen, was die Diagnose schwieriger macht, weil mehrere Alternativen zu berücksichtigen sind.

Subkategorisierungsfehler: Diese Fehler enstehen bei der Verletzung des Subkategorisierungsrahmens einer Kategorie, im allgemeinem bei Verben. In Semf untersuchen wir solche Restriktionsverletzungen ausschließlich bei Verben.

Morphologische Fehler

Morphologische Fehler bilden eine andere große Fehlerklasse, deren Bereich sich wiederum in vielen Fällen mit dem Bereich anderer Kategorien von Fehlern überlappt.

Wir haben uns im Rahmen dieser Fehlerklasse auf einige Flektionsprobleme und Aspekte des Komparativs (z.B. Konstruktionen der Form *big than*) beschränkt.

Orthographische Fehler

Aus Zeitmangel ist es uns nicht gelungen, eine richtige orthographische Fehleranalyse-Komponente zu implementieren. Die geringe Anzahl behandelter orthographischer Fehler beschränkt sich auf *false-friends*-Phänomene: es handelt sich hierbei um orthographische Interferenzen, die dadurch entstehen, daß in zwei verschiedenen Sprachen gewisse Wortformen große Ähnlichkeit aufweisen (**pyramide* statt *pyramid*).

Orthographische Fehler werden im Lexikon durch Fehlereinträge abgedeckt, daher soll auf diese Fehlerklasse hier nicht weiter eingegangen werden (siehe Kapitel 7).

11.2.3 Die Rolle von Grammatik und Fehlergrammatik bei der Fehlerbehandlung

Grammatik und Fehlergrammatik befinden sich zwar in separaten Datei-Modulen, trotzdem sind sie konzeptuell schwierig auseinanderzuhalten, da beide denselben Formalismus benutzen und in beiden Modulen Fehler behandelt werden. Was die Unterscheidung noch schwieriger macht, ist die Tatsache, daß einige Fehlerklassen, die mit Hilfe beider Systemkomponenten erkannt und beschrieben werden können, sich sogar noch überlappen.

Obligatorische Fehlereinträge und optionale Fehlereinträge

In Semf unterscheiden wir zwei Arten, auf die Fehler erkannt und beschrieben werden können. Zunächst existieren reine Fehlerregeln, die fehlerhafte Strukturen repräsentieren und nur im Fehlerfalle angewandt werden. Die in diesen Regeln enthaltenen Fehlereinträge sind obligatorisch, sie werden bei Anwendung der Regel immer in den Fehlerbaum aufgenommen. Unser Formalismus erlaubt aber auch optionale Fehlereinträge, die in Regeln für korrekte sprachliche Konstruktionen enthalten sein können. Solche Fehlereinträge werden nur dann im Fehlerbaum abgelegt, falls die durch das Weglassen von Anwendungsbedingungen abgewandelte Regel Anwendung gefunden hat, die damit zur Fehlerregel geworden ist. Zum Mechanismus, der das Weglassen solcher Bedingungen (*constraint relaxation*) und das Hinzutreten der optionalen Fehlereinträge regelt, siehe Kapitel 5. An dieser Stelle soll es genügen darauf hinzuweisen, daß auf diese Weise auch mehrere Fehler durch eine Regel diagnostiziert werden können.

Innerhalb der Grammatik gibt es nur optionale Fehlereinträge, während alle Regeln der Fehlergrammatik *genau einen* obligatorischen Fehlereintrag und möglicherweise optionale Fehlereinträge beinhalten. In Semf ist dies der prinzipielle Unterschied zwischen einer Grammatik-Regel und einer Fehlerregel in der Fehlergrammatik. Dieser Unterschied äußert sich allerdings auch in den mit beiden Mechanismen behandelbaren Fehlertypen.

Aufteilung der Fehlerklassen auf die beiden Module

Wie schon erwähnt, können in Semf mehrere gleichzeitig auftretende Fehler erkannt und beschrieben werden – dies ist sowohl durch mehrere Regeln möglich, als auch innerhalb einer einzigen.

Durch die Größe des Sprachfragmentes ist die Anzahl der möglichen Fehler verhältnismäßig klein und die möglichen Kombinationen dieser Fehler sind im allgemeinen überschaubar. So können in vielen Regeln gewisse, antizipierte Kombinationen von möglichen Fehler angegeben werden.

Obwohl diese Fehlerkombinationen überschaubar sind, ist ihre Anzahl ziemlich groß, was sich auf die Anzahl Fehlerregeln auswirkt. Unser Formalismus erlaubt es jedoch, mehrere Regeln zu einer einzigen Regel mit optionalen Einträgen zusammenzufassen. So werden für eine Regel alle Kombinationen der möglichen Fehler durch optionale Fehlereinträge abgedeckt. Diese enthalten dabei die einzelnen möglichen Fehlerarten, die Kombination wird durch das Zusammenwirken mehrerer solcher optionaler Einträge erreicht. Diese Eigenschaft unserer Fehlerbehandlung führt dazu, daß mit den optionalen Fehlereinträgen auch mit Hilfe von Grammatikregeln Fehler erkannt werden können.

Als Beispiel für solche Fehler sei der Kongruenzfehler genannt: Dieser äußert sich in der Verletzung einer Bedingung an die Regelanwendung, die beteiligten Konstituenten oder ihre Anordnung ändern sich dabei nicht. Bei Wortstellungsfehlern hingegen verändert sich die Anordnung der Konstituenten im Regelkopf, bei Satzkonstruktionsfehlern die Zusammensetzung. Bei diesen letzteren Fehlerarten ist deswegen die Aufstellung einer eigenen Fehlerregel notwendig, während Kongruenzfehler durch den Mechanismus der *constraint relaxation* im Rahmen der eigentlichen Regel behandelt werden können.

Würden keine Fehlereinträge in der Grammatik zugelassen, so müsste für den Kongruenzfehler in der Fehlergrammatik eine zusätzliche Regel enthalten sein, die denselben Regelkopf hat wie die dazu äquivalente Regel der Grammatik. Wir ersparen uns also mit der Aufteilung der Fehlertypen eine beträchtliche Anzahl zusätzlicher Fehlerregeln, darüberhinaus ist es auf dieselbe Weise möglich, einen gleichzeitigen Wortstellungs- und Kongruenzfehler innerhalb der Wortstellungsfehlerregel zu behandeln.

Alle so durch das Weglassen von Bedingungen und optionale Fehlereinträge implizit vorhandenen Fehlerregeln werden bei der Kompilierung der Grammatik als zusätzliche Fehlerregeln erzeugt, die die bisherige Regel ersetzen, tatsächlich findet also beim Parsing keine *constraint relaxation* statt. Jedoch müssen eine große Anzahl von Regeln nicht von Hand geschrieben werden, was einen riesengroßen Aufwand an Schreibarbeit und eine große Menge Programmcode erspart und zusätzlich dem System eine gewisse Eleganz verleiht.

Angesichts dieser Tatsachen wurden die behandelten Fehlerklassen auf die beiden Module Grammatik und Fehlergrammatik folgendermaßen aufgeteilt:

Kongruenz-, Rektions- und Morphologiefehler: Diese Fehlerklassen verlangen keine speziellen Regelköpfe und treten oft, sowohl alleine als auch in Kombination mit anderen Fehlerklassen, auf. Sie werden deshalb durch *constraint relaxation* behandelt und erscheinen sowohl in der Fehlergrammatik als auch innerhalb der Grammatik.

Satzkonstruktions- und Wortstellungsfehler: Diese Fehlerklassen können nicht innerhalb der Grammatik behandelt werden, da die dazugehörigen Regeln spezielle Regelköpfe besitzen, die in der Grammatik nicht auftreten können. (Im Falle von Konstruktionsfehlern sind Regelköpfe mit fehlenden oder überflüßigen Konstituenten notwendig, bei Wortstellungsfehlern werden Konstituenten vertauscht.)

Subkategorisierungsfehler: Obwohl Subkategorisierungsfehler, genauso wie Rektions-, Kongruenz- und Morphologiefehler, durch *constraint relaxation* behandelt werden, treten sie nur in der Fehlergrammatik auf, da sie normalerweise besonders grobe Regelverletzungen ausmachen und sich oft keine semantische Struktur bilden läßt, so daß im Gegensatz zur korrekten Regel die semantischen Koreferenzen irrelevant sind und sogar weggelassen werden müssen, um eine weitere fehlerhafte semantische Analyse oder sogar einen Abbruch der Verarbeitung zu verhindern.

Zusammenfassend werden also Kongruenz-, Rektions- und Morphologiefehler in beiden Modulen behandelt. Subkategorisierungsfehler, Wortstellungsfehler, Satzkonstruktionsfehler und – bedingt durch den in unserer Grammatik gewählten Umgang mit komplexen Präpositionen – teilweise lexikalische Fehler werden nur mit der Fehlergrammatik analysiert.

Fehlerkorrektur

Die obengenannten Fehlerklassen werden innerhalb der beiden Module zwar erkannt und relativ detailiert beschrieben, doch wird keine Korrektur durchgeführt, da sich das als ein schwerwiegendes Problem erwiesen hat und von Anfang an kein Schwerpunkt des Projekts war.

Die einzig vorgenommene Art von Korrektur ist, im Falle eines Kongruenzfehlers die Quantifizierung der jeweils als falsch angenommenen Konstituente in der semantischen Repräsentation an die der als richtig angenommenen anzugleichen, da sonst bei der weiteren Verarbeitung Probleme entstehen. Auf diese Weise ist es auch möglich, durch den Vergleich beider Semantiken mit der Weltrepräsentation die im Falle einer Inkongruenz fehlerhafte Konstituente zu identifizieren.

11.2.4 Aufbau des `ERROR`-Features

Das `ERROR`-Feature ist, zusammen mit den `SYN`- und `SEM`-Features, eine der drei Hauptmerkmalsstrukturen der Regeln. Innerhalb der `ERROR`-Strukturen wird die gesamte Information für die auf der „syntaktischen“ Ebene auftretenden Fehlerphänomene gespeichert. Im Verlauf eines Parses wird mit Hilfe dieses Features eine Baumstruktur erzeugt, deren Knoten auf Listen von Fehlereinträgen verweisen (die Listen sind leer, falls keine Fehler aufgetreten sind). Der Fehlerbaum wird ähnlich dem Syntaxbaum des Satzes aufgebaut, dabei werden die `ERROR`-Features der einfachen Konstituenten als Subfeatures durchgereicht. Die Listenstrukturen sind notwendig für gleichzeitig von einer Regel erkannte Fehler.

Technische Merkmale im ERROR-Feature

Die folgenden Merkmale sind im ERROR-Feature nur aus technischen Gründen enthalten. Mit ihrer Hilfe werden der Fehlerbaum erzeugt und die Fehlerbeschreibungen gesammelt. Sie dienen nicht der Fehlerbeschreibung.

`ERROR:` Das ERROR-Feature selbst ist vom Typ *subfeature*, bietet also die Möglichkeit, verschachtelte Strukturen aufzunehmen.

`ERR_1, ERR_2` sind ebenfalls vom Typ *subfeature* und diejenigen Merkmale, die den Baum aufspannen. Das ERROR-Feature einer Konstituente beinhaltet in ERR_1 und ERR_2 alle in ihren Teilkonstituenten aufgetretenen Fehler. Weil auf der rechten Seite des Kopfes einer Grammatik- oder Fehlergrammatikregel in der Semf-Grammatik nicht mehr als zwei komplexe Konstituenten auftauchen, ist hier die Beschränkung auf zwei Subfeatures möglich. Bei einer Erweiterung der Grammatik wären eventuell weitere notwendig.

`ERR_FIRST, ERR_NEXT:` Da durch eine Fehlerregel gegebenfalls mehrere Fehler erkannt werden können, mußte eine Listenstruktur eingeführt werden. Die Merkmale ERR_NEXT und ERR_FIRST (beide vom Typ *subfeature*) bilden diese Listenstruktur, wobei ERR_FIRST auf das erste Listenelement und ERR_NEXT auf den Rest der Liste zeigt.

`COUNT` ist ein Merkmal vom Typ *counter*. Dieses Merkmal berechnet die Anzahl der Fehler in einer Syntaxanalyse. Dabei addiert sich der COUNT-Wert jeweils aus den COUNT-Werten der Teilkonstituenten und der Anzahl der durch die aktuelle Regel erkannten Fehler.

Merkmale für die Fehlerbeschreibung im ERROR-Feature

Folgende Merkmale werden in Semf für die Fehlerbeschreibung benutzt:

`DESCRIPTION` ist ein Merkmal vom Typ *strdef*. Es enthält eine etwas kryptische Form von Fehlerbeschreibung. Ein Beispiel dafür ist:

```
DESCRIPTION = Syn_Subcat_Vbar_VpVbarAdv;
```

Das erste Wort im String gibt die Fehlerklasse an (in diesem Falle ist es ein syntaktischer Fehler), die nächsten Stringelemente spezifizieren den Fehler näher. Das zweite Wort bezeichnet die Fehler-Unterklasse (es handelt sich im Beispiel um ein Subkatekorisierungsfehler). Weiterhin wird die fehlerhafte Konstituente angegeben (VBAR), und schließlich die angewendete Regel (VP $\rightarrow$ VBAR ADV). Das DESCRIPTION-Feature findet bei der Fehlerverbalisierung Verwendung.

`TYPE` ist ein Merkmal vom Typ *attrdef* und spezifiziert die Fehlerklasse des aktuellen Fehlers (im Falle syntaktischer Fehler wird die jeweilige Subklasse angegeben). Dieses Merkmal wird nur intern benutzt.

`ERRORCAT` ist aus historischen Gründen vom Typ *attrlistdef* und gibt die fehlerhafte Konstituente an. Im Falle eines Wortstellungsfehlers werden allerdings nicht die vertauschten Konstituenten, sondern die übergeordnete Konstituente angegeben.

`ERRORNUM` ist vom Typ *attrdef* und gibt im Falle von Kongruenzfehlern den Numerus der als falsch erkannten Konstituente an. Der fehlerhafte Numerus wird durch eine Funktion berechnet, die im folgenden Abschnitt über Hilfsfunktionen ausführlich dargestellt wird.

`POSITION` ist ein Merkmal vom Typ *strdef.* Dieses Attribut ist ein String, in dem die Anfangs- und Endposition der fehlerhaften Konstituente innerhalb des Eingabesatzes gespeichert werden. Wie bei `ERRORNUM` werden die beiden Satzpositionen durch eine Funktion berechnet, doch in diesem Fall muß die Funktion auf Informationen aus der Chart zurückgreifen.

`PROBABILITY` ist ebenfalls als *strdef* definiert und enthält die Wahrscheinlichkeit, bzw. das *relative Gewicht* einer Fehlerhypothese. Auf der Basis der einzelnen Gewichte werden schließlich die Gesamtwahrscheinlichkeiten der alternativen Analysen berechnet, um die Parses in eine Reihenfolge bringen zu können. Die gewählten Werte sind rein intuitiv.

`INTERFERENCE:` Dieses Merkmal ist ein Flag, das auf Interferenz als mögliche Ursache des aktuellen Fehlers hinweist. Es ist vom Typ *attrdef* und kann die booleschen Werte YES und NO annehmen. Diese Werte werden bei der Fehlerverbalisierung berücksichtigt.

`GRAVITY` ist ein Merkmal vom Typ *attrdef* und wird nur im Falle schwerwiegender Fehler benutzt. Solche Fehler lassen die Erstellung einer Satzsemantik nicht zu. In Regeln, in denen das `GRAVITY`-Feature mit dem Wert `GRAVE` auftritt, entfallen deswegen die semantischen Koreferenzen. Die semantische Verarbeitung des Satzes entfällt in diesem Falle.

Hilfsfunktionen

Eine mächtige, allerdings wenig genutzte Eigenschaft unseres Formalismus, insbesondere der Fehlerbehandlung in Semf, ist die Möglichkeit, innerhalb der Grammatik (bzw. Fehlergrammatik) in C geschriebene Funktionen aufzurufen. Wir unterscheiden zwei Arten von Funktionen:

Testfunktionen treten im Regelkopf auf und erhalten (implizit) als Argument eine Liste mit den Strukturen aus `$0`, `$1`, ..., `$n`. Diese Funktionen können anhand beliebiger Kriterien darüber entscheiden, ob die aktuelle Regel angewandt werden soll oder nicht.

Funktionale Ausdrücke sind einfache Berechnungsfunktionen. Sie treten im Rumpf der Regel auf, als Argumente werden beliebig viele Pfadnamen explizit angegeben. Mit Hilfe dieser Funktionen werden Attributswerte berechnet.

Bei der syntaktischen Fehlerbehandlung werden vier Hilfsfunktionen benutzt:

- Die *checkEverything*-Funktion gehört als einzige zum ersten Funktionstyp und befindet sich in der Datei `fehlerbewertung.c`. Die Funktion wird aktiviert, falls eine Fehlerregel (ausschließlich der Kongruenzfehlerregeln) aktiviert werden soll. Sie überprüft rekursiv, ob in den Strukturen der Teilkonstituenten der aktuellen Regel Fehler vom selben Typ wie der aktuelle Fehler[3] vorgekommen sind. Ist dies der Fall, wird die Regel nicht angewandt und die aktuelle Parse-Alternative abgebrochen.

Die *checkEverything*-Funktion wird benutzt, um inkorrekte Fehleranalysen des Parsers zu verwerfen, die z.B. in einem korrekten Satz mehrere, einander aufhebende Wortstellungsfehler vermuten.

- Die *getNumerus*-Funktion gehört zur zweiten Kategorie von Funktionen; sie berechnet im Falle eines Kongruenzfehlers den Numerus der fehlerhaften Kategorie. Die Funktion erhält als Argument die Merkmalsstruktur der fehlerhaften Kategorie und holt sich daraus den Wert des `NUM`-Features; dieser Wert wird dann Attributswert von `ERRORNUM`.

- Die *getPosition*-Funktion ist ebenfalls eine Berechnungsfunktion, jedoch greift sie nicht auf Informationen aus der Merkmalsstruktur einer Teilkonstituente der Regel zurück, sondern direkt auf Informationen aus der Chart bezüglich der Position einer Konstituente im Eingabesatz. Die *getPosition*-Funktion schreibt die Anfangs- und Endposition der in der aktuellen Regel als fehlerhaft erkannten Konstituente in einem String.

- Die *modifyRange*-Funktion fällt auch unter die zweite Kategorie. Sie tritt aber nicht innerhalb des `ERROR`-Features auf, sondern nur im semantischen Teil einer Regel. Sie wird im Falle von Kongruenzfehlern zwischen Artikel und Nomen benutzt, um den Quantor der fehlerhaften Konstituente innerhalb des `SEM`-Features zu ändern. Diese Funktion ist daher nur auf semantischer Ebene relevant und wird im Kapitel 6.2.2 näher erklärt.

Die Möglichkeit, C-Funktionen innerhalb der Grammatik und Fehlergrammatik zu verwenden, die während der Laufzeit des Parsers Attributwerte berechnen oder Bedingungen überprüfen können, ist eine mächtige und sehr effiziente Methode, Informationen aus komplexen Merkmalsstrukturen zu gewinnen und redundante Parse-Alternativen zu verwerfen. Leider wurde dieses Instrument unseres Formalismus nur in geringem Maße benutzt; nichtsdestotrotz eröffnet dies eine Vielzahl von Möglichkeiten zur Erweiterung unserer Fehlerbehandlung.

11.2.5 Ein paar Beispiele von Regeln

Zur Veranschaulichung der im Kapitel erläuterten Eigenschaften der syntaktischen Fehlerbehandlung sollen in diesem Abschnitt ein paar relevante Beispiele von Regeln und Fehlerregeln diskutiert werden. Der semantische Teil der Regeln wird dabei weggelassen, da er keinen Schwerpunkt dieses Kapitels darstellt. Koreferenzen innerhalb des `SYN`- und `ERROR`-Features, die sich entweder wiederholen oder für unser Thema irrelevant sind, werden zur besseren Übersichtlichkeit ebenfalls nicht berücksichtigt.

[3] Der aktuelle Fehler ist der von der Fehlerregel erkannte Fehler.

Beispiele von Grammatikregeln

Als Beispiel für die Verwendung des ERROR-Merkmals in normalen Grammatikregeln sei *Beispielregel 1* gegeben, eine einfache Grammatikregel ohne optionale Fehlereinträge. Es handelt sich hier um eine Regel zur Erweiterung einer NBAR-Konstituente mit einer Präpositionalgruppe.

```
NBAR : NBAR PP [
       $0: [
            SYN = [ ... ]
            SEM = [ ... ]
            ERROR = [
                      ERR_1 = $1:ERROR;
                      ERR_2 = $2:ERROR;
                      COUNT = 0;
                      COUNT = $1:ERROR->COUNT;
                      COUNT = $2:ERROR->COUNT;
            ]         ]
       $1: [ ... ]
       ]
```

Regelbeispiel 1

Die ERROR-Features der NBAR ($1) und PP ($2) werden in ERR_1 und ERR_2 gespeichert. COUNT muß initialisiert werden: Da in $0 kein Fehler aufgetreten ist, wird es zunächst auf Null gesetzt. Die Inhalte der COUNT-Features der NBAR und der PP werden nachträglich addiert.

Beispielregel 2 zeigt eine komplexere Grammatikregel. Diese Regel beschreibt die Bildung einer Nominalphrase aus einem Artikel und einer NBAR-Konstituente. Dabei können Kongruenzfehler auftreten, die durch *constraint relaxation* behandelt werden.

```
AGR1|AGR2
NP : DET NBAR $$- * {*checkEverything} [
     $0: [
          SYN = [
                 @ AGR1 { NUM = $1:SYN->NUM; }
                 @ AGR2 { NUM = $2:SYN->NUM; } ...
                 ]
          ERROR = [ ...
                   COUNT = $c;
                   COUNT = $1:ERROR->COUNT;
                   COUNT = $2:ERROR->COUNT;
                   $$- AGR1 {
                     ERR_FIRST = [
                       DESCRIPTION = Syn_Agr_Num_Det_NpDetNbar;
                       TYPE = AGREEMENT;
                       ERRORCAT = DET;
                       POSITION = getPosition($1:SEM);
                       INTERFERENCE = NO;
                       PROBABILITY = 75%;
                       ERRORNUM = getNumerus($1:SYN->NUM);
                                 ] }
                     ERR_NEXT = [
                       $$- AGR2 {
                         ERR_FIRST = [
                           DESCRIPTION = Syn_Agr_Num_Nbar_NpDetNbar;
                           TYPE = AGREEMENT;
                           ERRORCAT = NBAR;
                           POSITION = getPosition($2:SEM);
                           INTERFERENCE = NO;
                           PROBABILITY = 50%;
                           ERRORNUM = getNumerus($2:SYN->NUM);
                                     ] } ...
          ]        ]             ]
     $1: [ ... ]
     $2: [ ... ]
     ]
```

Regelbeispiel 2

`AGR1` und `AGR2` bezeichnen die zwei sich gegenseitig ausschließenden optionalen Agreement-Constraints innerhalb des `SYN`-Features. Die anderen optionalen Bedingungen wurden aus Gründen der Übersichtlichkeit weggelassen.

Da es sich hier um eine Grammatikregel handelt, gibt es nur optionale Fehlereinträge: Dies wir daran deutlich, daß beide Fehlerbeschreibungen mit einem Label (`$$- AGR1` bzw. `$$- AGR2`) versehen sind, jeweils mit Bezug zu einem der beiden Constraints. Das Label `$$- *` in der Kopfzeile bewirkt, daß nur dann die Funktion *checkEverything* aufgerufen wird, wenn eines der beiden Constraints weggefallen ist.

Aus dieser Regel werden beim Kompilieren mehrere Regeln erzeugt: Eine davon ist die Grammatikregel (d.h. das `ERROR`-Feature enthält keinen Fehlereintrag), die anderen sind die mit je einem fehlenden Constraint und einem Fehlereintrag aufgebauten Regeln, ganz gewöhnliche Fehlerregeln für Kongruenzfehler. Da nicht beide Constraints fehlen dürfen (der gegenseitige Ausschluß ist durch die Disjunktion vor dem Regelkopf angegeben (`AGR1|AGR2`) können höchstens zwei Fehlerregeln generiert werden. Diese Fehlerregeln werden nur dann aktiviert, wenn tatsächlich ein Kongruenzfehler aufgetreten ist.

`COUNT` kann unterschiedliche Werte annehmen: In diesem Falle 0 (in der Grammatikregel) oder 1 (in den Fehlerregeln). Die Variable `$c` wird mit dem jeweiligen Wert belegt.

Die Wahrscheinlichkeit der beiden Fehler wird ziemlich hoch eingeschätzt, wobei es immer noch wahrscheinlicher ist, daß der Artikel falsch ist, als daß das Nomen im falschen Numerus steht.

Die Argumente der beiden Berechnungsfunktionen *getNumerus* und *getPosition* beziehen sich jeweils auf die fehlerhafte Konstituente, dabei tauchen als Argumente das `SYN->NUM`-Merkmal und aus implementatorischen Gründen das `SEM`-Merkmal der jeweiligen Konstituente auf.

Beispiele von Fehlerregeln

Als erstes sei eine einfache Fehlerregel gegeben, die für die Behandlung von fehlerhaften Konstruktionen wie etwa im Satz **The block is stands on the floor.* verwendet wird. Der Fehler kann sowohl morphologisch, als auch ein Satzbaufehler (fälschliches Hinzufügen eines Hilfsverbs) sein. Mit der *Beispielregel 3* wird der Satzbaufehler erkannt und beschrieben.

```
VBAR : AUX1 VBAR *checkEverything [
       $0: [
            SYN = [ ... ]
            SEM = [ ... ]
            ERROR = [
                     ERR_1 = $2:ERROR;
                     COUNT = 1;
                     COUNT = $2:ERROR->COUNT;
                     DESCRIPTION = Syn_Const_Extra_Aux_VbarAuxVbar;
                     TYPE = CONSTRUCTION;
                     ERRORCAT = AUX;
                     POSITION = getPosition($1:SEM);
                     INTERFERENCE = NO;
                     PROBABILITY = 30%;
            ]        ]
       $2: [ ... ]
       ]
```

Regelbeispiel 3

Die Regel bezieht sich auf eine mit einem Hilfsverb fehlerhaft erweiterte `VBAR`-Konstituente. Die *checkEverything*-Funktion wird hier zwingend aufgerufen.

Diese Fehlerregel bleibt nach dem Kompilieren unverändert: Es werden keine anderen Regeln daraus generiert.

Die Fehlerregel enthält einen obligatorischen Fehlereintrag: Wenn sie angewandt wird, wird dieser Eintrag automatisch im Fehlerbaum abgelegt. Der Fehlereintrag steht ist nicht mit einem Constraint-Label versehen und steht auch nicht zwischen geschweiften Klammern. Da mit Hilfe dieser Regel nur ein einziger Fehler erkannt werden kann, ist es außerdem nicht notwendig, innerhalb des `ERROR`-Features eine Listenstruktur zu benutzen.

`AUX1` ist eine einfache Kategorie, deshalb kann in `$1` noch kein Fehler aufgetreten sein. `ERR_1` erhält daher als Wert das `ERROR`-Feature der `VBAR` (`$2`).

`COUNT` wird auf 1 gesetzt, danach wird wiederum nur der Inhalt des `COUNT`-Features der `VBAR` dazu addiert.

Die Wahrscheinlichkeit des Satzkonstruktionsfehlers ist niedrig angesetzt, da der äquivalente morphologische Fehler als wahrscheinlicher anzusehen ist.

```
NBAR : VP NBAR *checkEverything #ORDER1 [
       $0: [
            SYN = [ ... ]
            SEM = [ ... ]
            ERROR = [
                     ...
                     ERR_FIRST = [
                       DESCRIPTION = Syn_Order_Inv_NbarVp_NbarVpNbar;
                       TYPE = WORDORDER;
                       ERRORCAT = NBAR;
                       POSITION = getPosition($0:SEM);
                       INTERFERENCE = NO;
                       PROBABILITY = 25%;
                                  ]
                     ERR_NEXT = [
                       $$- MORPH {
                         ERR_FIRST = [
                           DESCRIPTION = Morph_Infl_Form_Vp_NbarVpNbar;
                           TYPE = MORPHOLOGY;
                           ERRORCAT = VP;
                           POSITION = getPosition($1:SEM);
                           INTERFERENCE = NO;
                           PROBABILITY = 75%;
                                      ] }
            ]        ]            ]
       $1: [
            SYN = [
                   @ MORPH { FORM = PRESPART; }
                   ...
            ]      ]
       $2: [ ... ]
       ]
```

Regelbeispiel 4

Beispielregel 4 beschreibt eine Vertauschung zwischen einer Verbalphrase und einer NBAR-Konstituente. Die VP könnte dabei zusätzlich einen morphologischen Fehler beinhalten, der in die Listenstruktur des ERROR-Features als optionaler Fehlereintrag aufgenommen wird. Der zentrale Fehler ist aber der Wortstellungsfehler, der als obligatorischer Fehlereintrag auftritt.

Aus dieser Regel werden beim Kompilieren zwei Regeln generiert: Die eine enthält beide Fehlereinträge, die andere nur den Eintrag für den Wortstellungsfehler.

Das morphologische Constraint (MORPH) taucht hier innerhalb der $1-Struktur auf und wird vor den Unifikationen in $0 überprüft. Morphologische Fehler werden technisch wie Kongruenzfehler (durch Weglassen bestimmter Constraints) behandelt.

Das Label `#ORDER1` weist darauf hin, daß die Regel als *slave rule* zur Regelgruppe `ORDER1` gehört (d.h. wenn die *master rule* dieser Gruppe angewendet werden konnte, dann wird die vorliegende Regel blockiert.

Anmerkungen

Die vier Beispiele in diesem Abschnitt sind repräsentativ für die Implementation unserer Fehlerbehandlung und überdecken fast den ganzen Bereich der gebotenen Funktionalität. Innerhalb von Grammatik und Fehlergrammatik gibt es einige noch komplexere Regeln als die ausgewählten, ihre höhere Komplexität beruht jedoch nur auf einer größeren Anzahl von gleichzeitig auftretenden Fehlern und einer dadurch entstehenden tieferen Verschachtelung innerhalb der Fehlerliste.

Die Fehlertypen stellen ebenfalls eine repräsentative Auswahl dar, da Rektions- und letztlich Subkategorisierungsfehler wie Kongruenz- und Morphologiefehler behandelt werden, während Satzkonstruktionsfehler sich in der Behandlung wenig von Wortstellungsfehlern unterscheiden.

11.2.6 Schlußbemerkungen

Wie man sehen kann, ist die vom Parser durchgeführte Fehlerbehandlung ein verhältnismäßig komplexer Prozess, der die Mächtigkeit des benutzten Formalismus zur Geltung bringt.

Zusammenfassend beruht die Neuigkeit unserer syntaktischen Fehleranalyse auf den folgenden Eigenschaften:

- simultane Behandlung mehrerer nebeneinander auftretender Fehler
- Behandlung alternativer Fehlerhypothesen
- Zusammenfassung von mehreren Regeln zu einer einzigen Regel mit optionalen Einträgen, wodurch die Anzahl Regeln bzw. Fehlerregeln stark verringert wird
- Kontrolle über die Anwendung konkurrierender Regeln und Fehlerregeln
- Verwendung von C-Funktionen innerhalb von Grammatik und Fehlergrammatik

11.3 Semantische Fehlerbehandlung
Klaus Dalinghaus

Nachdem ein eingegebener Satz geparst worden ist und die aus den verschiedenen Resultaten entstandenen semantischen Repräsentationen von der Inferenzmaschine getestet wurden, ist es nun die Aufgabe, aus deren Ergebnissen abzuleiten, welche Interpretation nun die adäquateste ist, um dann darauf entsprechend reagieren zu können.

Die Grundidee der sog. Generalisierungs-Komponente besteht darin, von der gegebenen Repräsentation eines Satzes über die vorhandenen Informationen so zu generalisieren, daß sich eine Aussage ergibt, die in der Welt wahr ist. Dadurch kann man dann Schlüsse über die Wahrscheinlichkeit von Fehlern ziehen.

Beispiel: Der Schüler verwendet in seinem Satz die Nominalphrase (NP) "the green cubes", es gibt aber nur einen grünen Würfel in der Welt, jedoch existiert ein weiterer grüner Block. Man kann nun versuchen, vom Numerus *definit Plural* zu generalisieren, indem man den Numerus *indefinit Singular* wählt und dann den Satz noch einmal überprüft, diesmal mit Erfolg. Eine weitere Möglichkeit ist die Generalisierung von der Objektklasse *cube* zu der in der Hierachie eine Stufe höher stehenden Klasse *block*[4]. Die so entstandene neue NP hat nun zwei Bezugsobjekte in der Welt und damit den korrekten Numerus.

Als Informationen, für die eine Generalisierung möglich ist, können dabei herangezogen werden:

- Nominalphrase
  - Objektklasse: Anhand einer der Welt zugrunde liegenden Objekthierachie kann man stufenweise emporsteigen
  - Numerus: hier wird der Bereich der Zahl der erwarteten Objekte schrittweise erweitert.
  - Attribute: durch Weglassen kann von einzelnen Attributen abstrahiert werden.
- Verbgruppe: Durch die Verbgruppe wird eine Relation definert, z.B. durch *lying on*, anhand der man abstrahieren kann.

Aus zeitlichen Gründen habe ich mich bei der Implementation auf einige Aspekte der Nominalphrasen konzentriert.

Bevor jedoch Generalisierungen durchgeführt werden können, ist eine Analyse der Ausgabe der Inferenzmaschine notwendig. Es müssen dabei alle ermittelten Informationen gespeichert werden, damit eine Entscheidung getroffen werden kann, welche Anfrage auf welche Art bearbeitet werden soll. Als nächstes folgt dann die Auswahl des Generalisierungspunktes für eine Anfrage, d.h. die Stelle in einer Anfrage, an der das Ergebnis nicht mit dem gewünschten übereinstimmt und verbessert werden muß bzw. kann. Danach kann man versuchen, die einzelnen Schritte der Generalisierung durchzuführen, je nachdem, welche Klasse von Information man ausgewählt hat. Die so entstandene neue Repräsentation des Satzes kann als Anfrage wieder an die Inferenzmaschine zur erneuten Bearbeitung geschickt werden. Nachdem dessen Ergebnis erneut geparst wurde, kann man feststellen, ob sich an der geänderten Stelle ein Erfolg eingestellt hat.

[4]zur Hierachie von Objektklassen siehe Kapitel Weltrepräsentation

Im folgenden sollen die wichtigsten Punkte dieses Ablaufs noch einmal genauer beschrieben werden.

11.3.1 Analyse der Inferenzergebnisse

Im Analyseschritt sind alle nötigen Informationen aus den einzelnen Anfragen an die Inferenzmachine zu extrahieren und aufzubereiten. Hierzu werden zunächst alle Objektbeschreibungen in der Anfrage lokalisert. Jeder Objektgruppenbeschreibung ist ein eindeutiger Tag vorangestellt, um Referenzen bei der Inferenz von der Antwort auf die Anfrage erstellen zu können. Anschließend kann mit der Analyse der Antwort begonnen werden.

Die Verarbeitung einer Objektgruppenbeschreibung liefert eine Clusternummer, eine Id-Liste und evtl. eine Gruppen-Id[5]. Die Auswertung einer Relation erfolgt jeweils nach der Angabe der darin verwendeten Gruppen. Anschließend folgt eine Auswertung der Quantoren für die einzelnen Gruppen, wobei die prozentualen Angaben durch konkrete für die aktuelle Welt zutreffende Werte ersetzt werden. Die Auswertung liefert die Anzahl Objekte, die es in dieser Gruppe gibt und die Anzahl der Objekte, die davon die angegebene Relation erfüllen. Abschließend folgt dann eine Auswertung der Relation durch ein *ok* oder durch ein *fail*. Sofern weitere Relationen mit weiteren Objektgruppen vorhanden sind, wiederholt sich nun der Prozeß.[6]

Nach Abschluß der Analyse stehen also folgende Informationen zur Verfügung:

- Objektgruppe
  - Quantor: Range entpr. der Beschreibung in der Grammatik
  - Id's: Tag, Cluster[7], Gruppen-Id
  - Range: Informationen darüber, wieviele Objekte mindestens und höchstens in diese Klasse fallen dürfen gemäß dem Quantor, sowie über die Anzahl der existierenden und die Anzahl der davon die verlangte Relation erfüllenden Objekte.
  - String: Zugriff auf die Zeichenkette in der Anfrage[8] für evtl. Änderungen
- Relation
  - Name: Zugriff auf die Zeichenkette in der Abfrage (s.o.)
  - Zuordnung: Eine Liste von Objekttupeln, die auf die Relation getestet wurden mit dem jeweiligen Ergebnis.

[5] man erkennt dies am g in der Quantorinformation, d.h. über diese Gruppe von Objekten soll weiter als Individuum inferiert werden.

[6] Für eine detailiertere Beschreibung siehe Kapitel "Weltrepräsentation".

[7] incl. Cluster-Level, da in jeder Klammerebene der Anfrage die Numerierung wieder bei 0 beginnt

[8] über diese Zeichenkette ist auch die Entitätsbezeichnung der Gruppe zugreifbar

11.3.2 Auswahl des Generalisierungspunktes

Nachdem aus dem Ergebnis der Anfrage alle Informationen extrahiert worden sind, kann die Suche nach einer geeigneten Stelle, an dem sich die Druchführung einer Generalisierung lohnt, beginnen. Diese Stelle in der Anfrage soll im weiteren mit Generalisierungspunkt bezeichnet werden. Einen solchen Punkt findet man dort, wo eine Rangeforderung verletzt wird. Dies kann folgende Ursachen haben.

1. Die Anzahl der Objekte in einer Gruppe, die die geforderte Relation erfüllen, ist kleiner als die Anzahl der im Quantor minimal geforderten Objekte. Noch spezieller: Die Anzahl der existierenden Objekte ist zu klein.

2. Die Anzahl der Objekte in einer Gruppe, die die geforderte Relation erfüllen, ist größer als die Anzahl der im Quantor maximal geforderten Objekte.

Weiter muß man sich nun fragen, wie eine solche Verletzung der Rangeforderung zustande gekommen ist, um daraus dann einen Algorithmus für dessen Behebung abzuleiten. Sollten zuwenig Objekte in einer Gruppe sein, so kann dies an der Verwendung des falschen Quantors liegen. Dies ist allerdings eher unwahrscheinlich, wenn kein Objekt in der Gruppe ist. Eine weitere Lösung könnte die Spezifikation mit einem falschen Attribut sein, z.B. eine Farbverwechslung oder gar die Verwendung einer falschen Entität für die Objektgruppe, z.B. *cube* statt *block*. Sollte man jedoch genügend Objekte in der Gruppe haben, nur davon nicht genügend die Relation erfüllen, könnte man sich auch überlegen, die Relation so zu modifizieren, daß mehr Objekte diese erfüllen. Dabei kann die Modifikation einer Relation durchaus bedeuten, daß man eine Objektgruppe modifiziert. So könnte man bei einem Test der Relation r zwischen der Gruppe a und der Gruppe b, der scheitert, weil zu wenig Elemente aus der Gruppe a die Relation erfüllen, auch versuchen, die Anzahl der Elemente der Gruppe b zu erhöhen, um die Gültigkeit der Relation damit zu verbessern.

11.3.3 Durchführung der Generalisierung

In diesem Abschnitt werden sowohl die bereits implementierten Teile des Algorithmus als auch die aus Zeitgründen nicht berücksichtigten Teile beschrieben.

Sicherlich ist die nun folgende Liste der Ansatzpunkte für die Durchführung einer Generalisierung nicht vollständig. Ich habe mich dabei auf die implementierten, bzw. theoretisch bearbeiteten Punkte beschränkt.

Verwendung des falschen Quantors

Sollte die Anzahl der Objekte, die der Beschreibung genügen nicht im Bereich des in der Objektbeschreibung angegebenen Quantors liegen, so muß man die Fälle unterscheiden, ob die Obergrenze überschritten wurde oder die Untergrenze unterschritten wurde und ob die Grenze 1 ist.

Wurde die Obergrenze überschritten, so kann man sicherlich die Obergrenze weiter nach oben verschieben, um dieses Problem zu lösen. Allerdings stellt sich hier die Frage, ob die Untergrenze dabei gleich bleibt. Wenn man beispielsweise aus einem Range von [*%50 %50*], was annähernd dem Quantor *half* entspricht, den Range nach oben erweitern will, so sollte man dies nur mit einer sprachlichen Anlehnung tun, etwa [*%51 %99*] für *most* könnte hier die naheliegende Lösung sein. Entsprechendes gilt für die Verschiebung der Untergrenze nach unten.

Sollte die Obergrenze *1* sein, so muß man sich darüber im Klaren sein, daß eine Erhöhung u.U. eine Wandlung der zugehörigen Nominalphrase vom Singular in den Plural bewirkt. Dies also dann Fehler im syntaktischen Bereich hervorrufen würde. Umgekehrt erhält man durch das Herabsetzen der Untergrenze von *1* auf *0* eine negierte Aussage[9], was in den meisten Fällen wohl nicht beabsichtigt gewesen sein dürfte.

Die Behandlung der Quantorengrenzen wurde nicht implementiert.

Überspezifikation einer Objektgruppe

Die Überspezifikation einer Objektgruppe kann ihre Ursache darin haben, daß falsche oder einfach nur zu viele Attributangaben gemacht wurden oder daß die Beschreibung der Entität nicht richtig gewählt wurde. Letzteres ist z.B. der Fall, wenn mehrere Objekte mit *cubes* bezeichnet werden sollen, obwohl einige davon keine Würfel, sondern Quader sind, die mit *blocks* zu bezeichnen wären.

Im Fall der Attributangaben wird versucht, einzelne Spezifikationen wie die Angabe einer Farbe aus der Beschreibung zu entfernen. Da die daraus resultierende Anfrage nicht notwendigerweise die Obergrenze im Quantor einhalten muß, wird diese auf *%100* erhöht. Die neue Anfrage wird von der Inferenzmaschine bearbeitet und das Ergebnis wieder analysiert. Erhält man nun genügend Objekte in der Beschreibung, so kann man sich die Werte aller dieser Objekte bezüglich des entfernten Attributes ansehen, um Schlüsse darüber zu ziehen, ob die Verwechselung des Wertes mit einem anderen einen Sinn ergibt. Dieser Vorgang ist jedoch nicht implementiert. Die Systemreaktion lautet einfach z.B. "Du hast die Farbe verwechselt." Sollte das Entfernen eines Attributes noch keine Erfolge aufweisen, wird zusätzlich versucht, mehrere Attribute wegzulassen.

Im Fall der falschen Verwendung der Entitätsbezeichnung wird die Bezeichnung durch die in der Hierachie der Welt eine Stufe höher stehenden Bezeichnung ersetzt. Ähnlich zum ersten Fall wird auch hier die Obergrenze des Quantors auf *%100* erhöht und anschließend das Ergebnis betrachtet. Auch dieses Verfahren kann man mehrmals anwenden, indem man immer weiter in der Hierachie der Entitäten nach oben geht. Die Systemreaktion heißt in diesem Fall "Du hast ein Objekt falsch benannt."

Sowohl bei den Attributen als auch bei der Entitätsbezeichnung ist die Anzahl der durchzuführenden Schritte sorgsam abzuwägen, da bei genügend großer Schrittzahl immer ein Generalisierungsergebnis eintritt, daß aber einen immer unbedeutenderen Wert hat, je mehr Stufen durchgeführt wurden. In einem "vernünftigen Rahmen" können beide Verfahren kombiniert werden.

In einigen Fällen kann es sinnvoll sein, eine Objektbeschreibung sogar ganz wegzulassen, z.B. in einer Koordination von mehreren Nominalphrasen.

[9]wie sich dadurch die Obergrenze verändert, soll hier offen bleiben.

Modifikation der Relation

Sollte beim Test des Ranges für eine Objektgruppe die Anzahl der Objekte, die die geforderte Relation nicht erfüllen, nicht hoch genug sein, jedoch die Anzahl der Elemente der Gruppe hoch genug sein, so kann dies natürlich auch an der Restriktion durch die Relation liegen. Hier kann man nach verschiedenen Ursachen forschen, die ein gewisses Wissen über die verwendete Relation und die damit "verwandten" Relationen vorraussetzt:

- Hat der Schüler die Relation verwechselt?

  Dies kann man durch Ersetzen der Relation druch die "Verwandten" prüfen, wobei Relationen nicht nur durch semantische Ähnlichkeit, wie z.B. *on* und *over*, sondern auch durch sprachliche Ähnlichkeit, wie z.B. "False Friends" bei *before* und *vor* verwandt sein können.

- Hat der Schüler die Argumente der Relation vertauscht?

  Bei Relationen, die nicht symmetrisch sind wie *over* oder *left* kann dieser Test durchaus bessere Ergebnisse bringen, die dann vielleicht auch auf ein falsches Verständis dieser Relation vom Schüler schließen lassen.

- Hat der Schüler die Relation zu stark relativiert?

  Bei Relationen wie *next to* gehen bei vorgegebenen Objekten die Meinungen bei verschiedenen Personen oft auseinander, ob die Relation nun gültig ist oder nicht. Man könnte dies überprüfen, indem man die angegebene Relation mit einer größeren Toleranz noch einmal prüft, um dem Schüler mitzuteilen, daß diese Aussage nicht falsch, aber für das System nur in einem entfernten Sinn gültig ist.

11.3.4 Bewertung der Ergebnisse

Nachdem die Generalisierung abgeschlossen ist, müssen die gefundenen Ergebnisse bewertet werden, damit bei mehreren verschiedenen Lösungen eine Entscheidung für den wahrscheinlichsten Fehler möglich wird. Aus zeitlichen Gründen ist dieser Punkt jedoch entfallen, so daß in Semf zunächst alle möglichen Fehler angezeigt werden.

11.4 Fehlerbewertung
Henrik Dittmann

Ein für das in *Semf* verfolgte Konzept wesentlicher Mechanismus, der jedoch nicht mehr implementiert werden konnte, beinhaltet die Integration der von syntaktischer und semantischer Fehlerbehandlung bereitgestellten Ergebnisse. Anhand der jeweiligen Fehlertypen und -wahrscheinlichkeiten soll eine Heuristik die insgesamt plausibelste Fehlerhypothese für die Rückmeldung an die Lernerin auswählen. Es ist auch denkbar, diese zusammen mit den in der Rangfolge nächsthöheren Hypothesen an eine Tutorkomponente weiterzuleiten.

Als Vorbedingung für eine Fehlerbewertung wären seitens der semantischen Fehlerbehandlung eine möglichst vollständige Beschreibung der Abweichungen von der Welt notwendig – die in Kapitel 11.3 beschriebenen Mechanismen sind nicht alle implementiert. Darüberhinaus muß diese Komponente eine Bewertung der Satzinterpretationen liefern, auf deren Basis die Fehlerbewertung arbeiten kann.

Rolle der Fehlerbewertungskomponente wäre es dann, mittels einer Heuristik, die zum Teil auf den von den beiden Fehlerbehandlungskomponenten gelieferten Wahrscheinlichkeitsannahmen basiert, eine Gesamtwahrscheinlichkeit zu ermitteln. Besondere Aufmerksamkeit muß dabei auch auf die jeweiligen Fehlertypen gerichtet werden. So wäre z.B. eine Syntaxanalyse, die einen fehlerhaften Determiner annimmt, bei einem von der semantischen Fehlerbehandlung erkannten Quantorenfehler wesentlich wahrscheinlicher als bei der Diagnose einer Attributverwechselung.

Da sich eine Systemreaktion sehr unterschiedlich gestalten kann und sollte, wäre eine eigenständige Tutorkomponente wünschenswert, die anhand von Kriterien wie der Schwere der Fehler, aktuellen Lernzielen und erkannten Schwächen der Lernerin eine geeignete Form der Rückmeldung (Sprung zu einer anderen Übung, sprachliche Fehlerbeschreibung, Ignorieren des Fehlers etc.) findet.

11.4.1 Die implementierte Rückmeldung

Die im *Semf*-Prototypen implementierte Rückmeldung besteht lediglich aus einer Komponente, die die erkannten Fehler und Abweichungen von der Welt mit Hilfe von Templates verbalisiert. Dies soll lediglich dazu dienen, die Funktionalität unseres Systems besser demonstrieren zu können. Alle Analysen des Eingabesatzes werden in der Reihenfolge präsentiert, die durch die syntaktische Fehlerbehandlung ermittelt worden ist.

Kapitel 12

Übung
Gerd Bongardt

Das Projekt Semf hatte von vorneherein eine eher forschungsorientierte Ausrichtung. Um neue Wege aufzuzeigen, sind deshalb die Schwerpunkte der Arbeit eher auf das Zusammenspiel zwischen den einzelnen Komponenten als auf die für den Lerner relevanten Aspekte gerichtet worden.

Die Übungswelten, die zum Schluß als Übungen über die WWW-Oberfläche sichtbar sind, haben deshalb auch keinerlei didaktischen Anspruch. Sie sind nur zu Demonstrationszwecken implementiert worden. Alle drei Szenen der Übungen unterscheiden sich deshalb nur durch die im Bild sichtbaren Objekte, bzw. deren Anordung. Für den Benutzer ist nur die Aufgabe *Please describe the scene above:* implementiert. Nach der Eingabe kann er nun den Input vom System bearbeiten und sich anschließend *Analyse (Syntax, Semantik, Fehlerinformationen), Fehlerbeschreibungen, Semantik, das Ergebnis der Inferenz, Verbalisierung, Generalisierung* oder die *nächste Szene* anzeigen lassen.

Als Übungen waren neben Szenenbeschreibungen auch Bewegungen (wie z.B. schiebe einen roten Block hinter einen grünen) vorgesehen. Dieses ist allerdings noch nicht implementiert. Erst die Möglichkeiten von Bewegungen würden sinnvolle Übungstypen zulassen. Jedoch zeigt schon das jetzige System, daß verschiedene Übungstypen sich mit ihm realisieren lassen. Vorstellbar wäre z.B. eine Graphik durch sprachliche Beschreibungen zu verändern oder neue Objekte einzufügen, bzw. zu entfernen. Auch ist eine aufgesetzte tutorielle Komponente, die durch die Fehlererkennung zu weiteren vertiefenden Übungen führt vorstellbar.

Semf war als kleine Teilkomponente eines CALL-Systems vorgesehen, dieser Umstand rechtfertigt, daß das Thema “Übungen” nur am Rande behandelt worden ist. Der jetzige Stand des Systems läßt durch den modularen Aufbau weitere Ergänzungen für sinnvolle Übungen zu.

Kapitel 13

Benutzeroberfläche

Georg Rehm

13.1 Einführung

Das Semf-System wird über eine Hypertext-Oberfläche bedient, die über das World Wide Web (WWW) im Internet – und somit weltweit – zur Verfügung steht.[1] Die Oberfläche besteht zur Zeit aus verschiedenen HTML-Seiten[2], auf denen unterschiedliche Blockswelten zu sehen sind, die der Schüler aufgefordert wird zu beschreiben. An dieser Stelle soll ausdrücklich darauf hingewiesen werden, daß diese Art der Übungsgestaltung keinen didaktischen Ansprüchen genügt. Vielmehr handelt es sich beim Semf-System um einen Forschungsprototypen, bei dem andere Aspekte des *Computer Assisted Language Learning* im Vordergrund stehen. Die Einbettung des Systems in einen didaktisch sinnvollen Übungskontext kann zu einem späteren Zeitpunkt, beispielsweise im Rahmen eines Nachfolgeprojekts oder einer Magisterarbeit, erfolgen. Primäres Ziel bei der Entwicklung dieses Front-Ends war eine möglichst intuitive Bereitstellung der Daten, die das Semf-System im Zuge der Analyse einer Eingabe produziert.

13.2 Implementation

13.2.1 Aufbau der Benutzeroberfläche

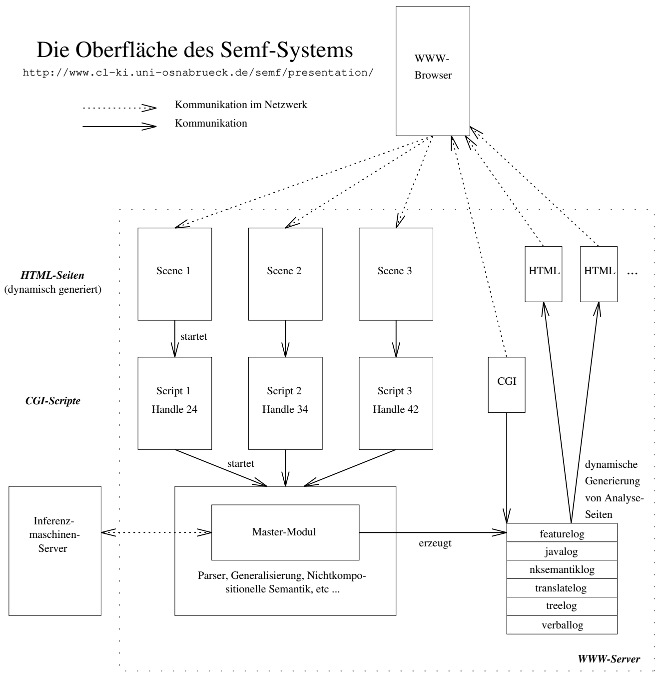


Abbildung: Architektur der Oberfläche

[1] Zu erreichen über die Adresse `http://www.cl-ki.uni-osnabrueck.de/semf/`

[2] HTML: Hypertext Markup Language, die Seitenbeschreibungssprache des World Wide Web

13.2.2 Die Übungsseite

Die HTML-Seite, die eine Übung konstituiert, besteht im wesentlichen aus einer zu beschreibenden Grafik, die natürlich auch als Szene – als Zustand der Welt – in der Weltrepräsentation vorhanden ist, einer Zeile, in der die Beschreibung eingegeben werden kann und einem Menü, aus dem man die verschiedenen aus der Analyse gewonnenen Daten wählen kann. Desweiteren kann mittels eines Knopfes eine rudimentär implementierte Anaphernresolution aktiviert werden.

Gibt der Schüler einen Satz ein und drückt den Knopf mit der Aufschrift *Process Input*, so wird ein CGI Skript[3] aktiviert, das den Analyseprozeß der Eingabe durch das Semf-System startet. Wichtig ist hierbei, daß beim Aufruf des Parsers das richtige Handle, die "Nummer" des Weltzustands, für die gezeigte Welt mit übergeben wird, damit ein korrekter Vergleich der vom Schüler eingegebenen Beschreibung und der aktuellen Konfiguration der Blockswelt stattfinden kann. Über ein sogenanntes *hidden field* wird von jeder der drei HTML-Übungsseiten aus eine Variable des CGI Skriptes mit dem jeweils korrekten Handle bedient.

Das CGI Skript wandelt die Schülereingabe zunächst in eine für die weitere Verarbeitung notwendige Form um, wobei alle Großbuchstaben in Kleinbuchstaben konvertiert werden. Daraufhin wird der Satz in eine Temporärdatei geschrieben, um ihn später als Überschrift (siehe Abbildungen) in die dynamisch generierten Analyseseiten einzubinden. Hierauf wird der Master gestartet, der wiederum den Parser mit dem Eingabesatz füttert. Der Parser schreibt nun nach der Analyse eine gewisse Menge Logdateien an einen dedizierten Platz im Filesystem des Webservers, die im Folgenden verarbeitet werden. Diese Logdateien stellen die Informationen dar, die im Verlauf der Verarbeitung von den einzelnen Moduln generiert wurden.

13.2.3 Die Aufbereitung der Logfiles

Nachdem nun die Logdateien bereitliegen, werden verschiedene Tools gestartet, die zwecks einer besseren Visualisierung einige Daten, die in ASCII-Form in den Logdateien enthalten sind, in das Grafikformat GIF[4] umwandeln. Dies geschieht mit der syntaktischen (`treelog`) und semantischen (`featurelog`) Struktur des Eingabesatzes sowie mit den Informationen über eventuell erkannte syntaktische Fehler. Die auf diese Weise erzeugten Graphiken werden nun in dynamisch generierte HTML-Seiten integriert, wobei für jeden Parse je eine Seite erstellt wird. Die restlichen Logdateien sind über CGI Skripte zugänglich, die zugleich eine Umwandlung der unformatierten ASCII-Daten nach HTML vornehmen.

[3]CGI: Common Gateway Interface; der Standard, um mit selbst entwickelten Programmen die Funktionalität eines WWW-Servers zu erweitern

[4]Graphics Interchange Format

Logfile	Inhalt	Dateityp
`treelog`	Syntaxbaum	GIF
`featurelog`	Semantik-Struktur und ggf. Fehlerbeschreibungen	GIF
`translatelog`	Fehlerbeschreibungen im Klartext	HTML
`nksemantiklog`	die konstruierten Semantiken	HTML
`javalog`	Ausgabe der Inferenzmaschine	HTML
`generallog`	Ausgabe der Generalisierung	HTML
`verballog`	Ausgabe der Fehlerverbalisierungen	HTML

Tabelle: Die verschiedenen vom Parser geschriebenen und von der Benutzeroberfläche verwendeten Logdateien

13.3 Die Oberfläche

Abbildung einer prototypischen Übung

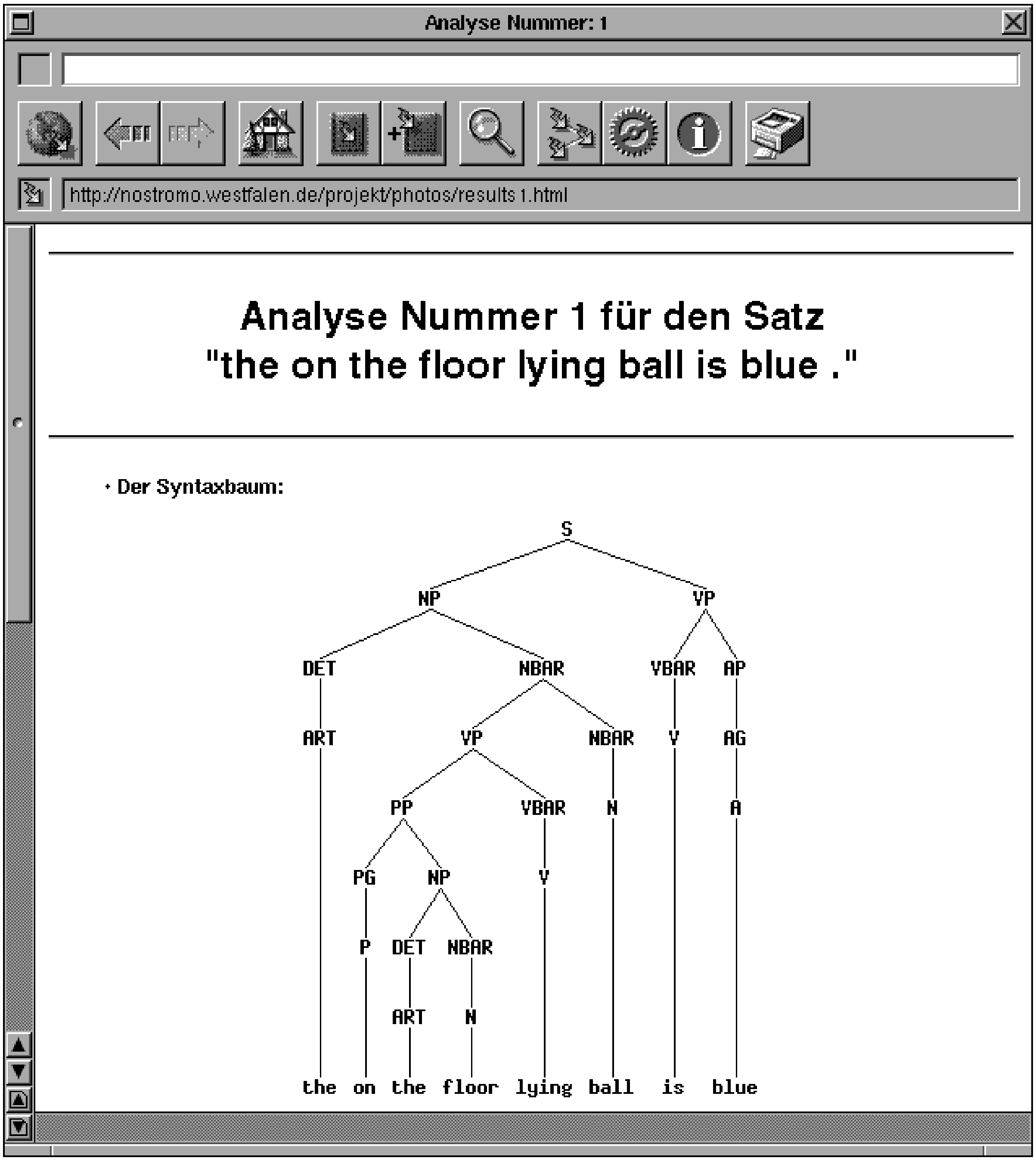


Abbildung: Der erzeugte Syntaxbaum

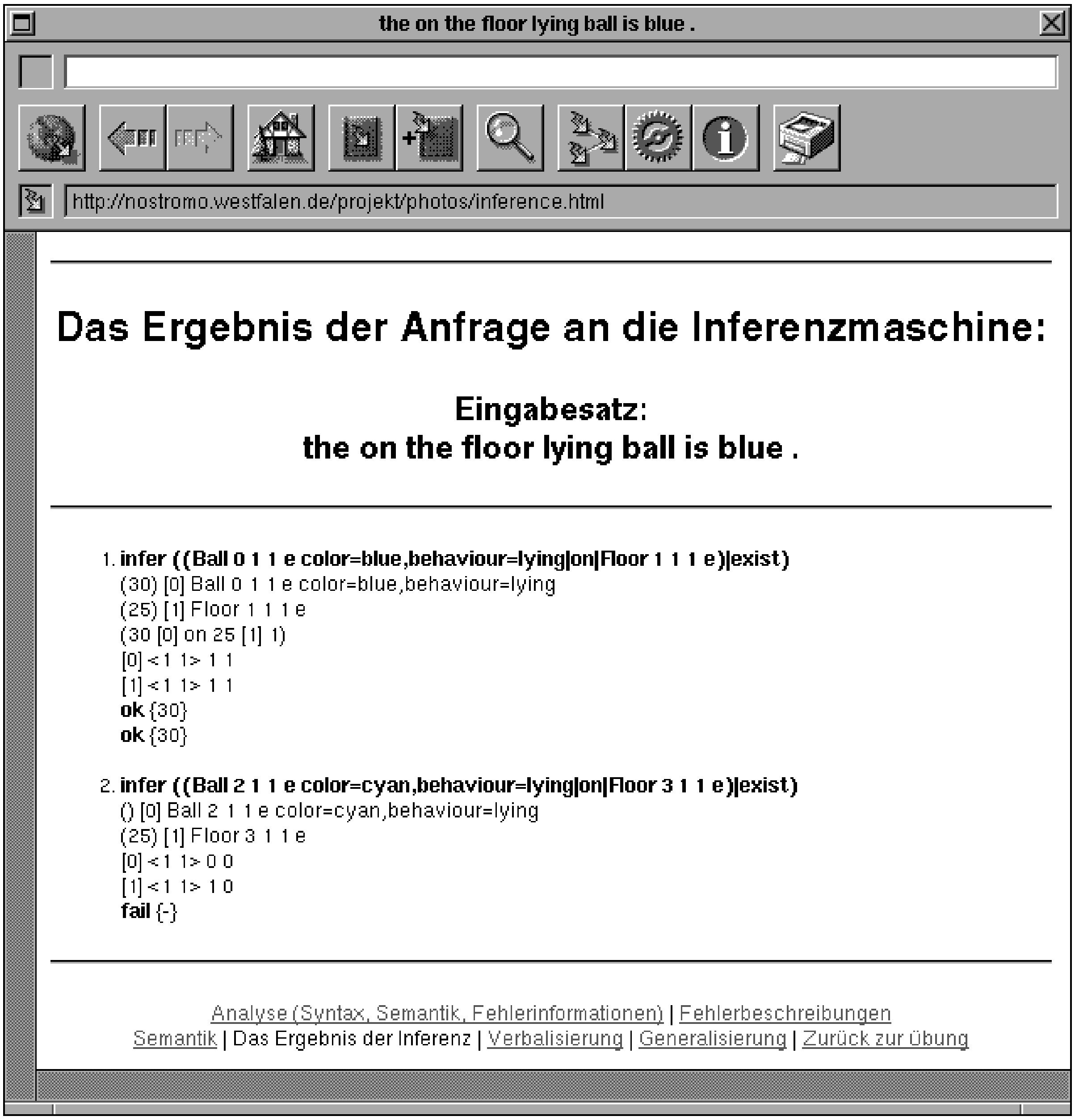


Abbildung: Die Ausgabe der Inferenzmaschine

13.4 Ausblick

Verschiedene Aspekte dieser Oberfläche sind verbesserungsbedürftig. Zum einen sollte das System, das auf einem herkömmlichen Web-Server läuft[5], multi-Zugriff sicher gemacht werden; bislang

[5] Software: `Apache 1.1.1`

kann nur *ein* gleichzeitiger Zugriff erfolgen, da die umgewandelten Logdateien an dedizierte Stellen im Dokumentenast des Servers geschrieben werden, damit es in den HTML-Dokumenten möglich ist, Links auf die dynamisch generierten Analyse-Seiten einzufügen.

13.5 Die Visualisierung des Parser-Outputs *Klaus Dalinghaus*

Die Ausgaben des Parsers erfolgen im ASCII-Format in Dateien. Da diese Ausgaben in linearer Form und ungefiltert gemacht werden, ist es ziemlich aufwendig, darin die verwendeten Regeln nachzuvollziehen oder zu prüfen.

Um eine bessere Übersicht über die Informationen des Parsers zu liefern, entstand das Tool *parview*, das in zwei Versionen vorliegt. In einer ersten Version wird die Ausgabe des Syntaxbaumes mit Hilfe eines Konverters[6] in TeX-Code umgewandelt, dann übersetzt und als PostScript auf dem Bildschirm angezeigt. Entsprechend wurde ein Programm entwickelt, daß die Feature-Term-Strukturen in TeX-Code umwandelt, damit auch diese als PostScript getrennt nach den Haupt-Features *SEM* für Semantik und *ERR* für Fehler auf den Bildschirm anzeigt werden können.

Da jedoch für etwas größere Strukturen die Umwandlung und vor allem die Übersetzung von TeX-Code ziemliche Performanzprobleme aufzeigten, und außerdem die Integration in die WWW-Oberfläche nicht zufriedenstellend war, entstanden mit Hilfe eines weiteren Tools[7] Programme, die die vorhandenen linearen Daten in GIF-Bilder mit den jeweiligen Strukturen (Syntaxbäume bzw. Feature-Term-Strukturen) umwandeln.

13.5.1 Syntaxbäume – “tree2gif”

Die Informationen über die enstandenen Syntaxbäume werden vom Parser in folgender Struktur[8] in das logfile geschrieben:

```
<Satz> ::= <Baum>

<Baum> ::= <Knoten> '(' <Baumliste> || <Wert> ')'

<Knoten> ::= <Nicht-Terminal-Symbol>

<Baumliste> ::= <Baum> || <Baum> <Baumliste>

<Wert> ::= <Terminal-Symbol>
```

[6]Es handelt sich dabei um das Programm tree von Greg Lee, das es erlaubt, Syntaxbäume sowohl als ASCII-Text als auch als TeX-Code auszugeben.

[7]Es handelt sich hierbei um die GD-Library, eine GIF-Manipulating Library von Tom Boutell, mit der man z.B. innerhalb eines C-Programmes GIF-Bilder erstellen, speichern, laden und bearbeiten kann.

[8]Legende siehe Sprachbeschreibung der Inferenzmaschine

Als Nicht-Terminal-Symbol und Terminal-Symbol sind beliebige Zeichenketten zugelassen, die mit einem Buchstaben beginnen, zur besseren Unterscheidung haben wir jedoch als Nicht-Terminal-Symbole nur solche mit Großbuchstaben gewählt.

Nach einem Parserdurchlauf kann man nun mit dem Tool "tree2gif" die entsprechenden Syntaxbäume als GIF-Bilder erstellen, wobei das Programm automatisch die Breite des Bildes errechnet.

Ein zusätzliches Feature besteht darin, daß von bestimmten, für den Betrachter eher verwirrenden, technischen Kategorien abstrahiert werden kann. So wird beispielsweise in der Grammatik die komplexe Präposition *on top of* mit Hilfe künstlich eingeführter Kategorien aus den einzelnen Wörtern *on*, *top* und *of* zusammengesetzt. Im vereinfachten Syntaxbaum wird dies jedoch weggelassen (siehe Beispiel).

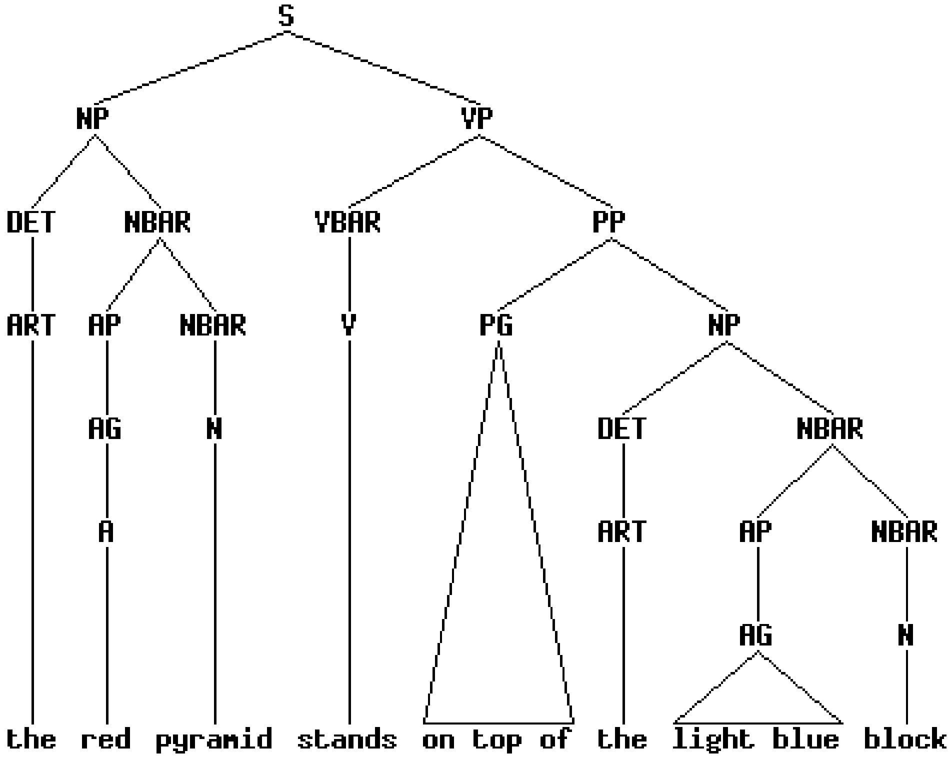


Abbildung: vereinfachte, graphische Darstellung eines Syntaxbaumes

13.5.2 Feature-Term-Strukturen – "feat2gif"

Die Informationen über die entstandenen Feature-Term-Strukturen werden vom Parser in folgender Struktur in das Logfile geschrieben:

```
<Satz> ::= <Feature-Liste>

<Feature-Liste> ::= <Feature> || <Feature> <Feature-Liste>
```

`<Feature> ::= <Feature-Name> '=' <Feature-Wert> ';'`

`<Feature-Wert> ::= <Wert> ‖ '[' <Feature-Liste> ']'`

`<Feature-Name> ::= <Symbol-Kette>`

`<Wert> ::= <Symbol-Kette>`

Hier gilt als Faustregel, daß als Feature-Namen nur Zeichenketten verwendet werden, die ausschließlich aus Großbuchstaben bestehen.

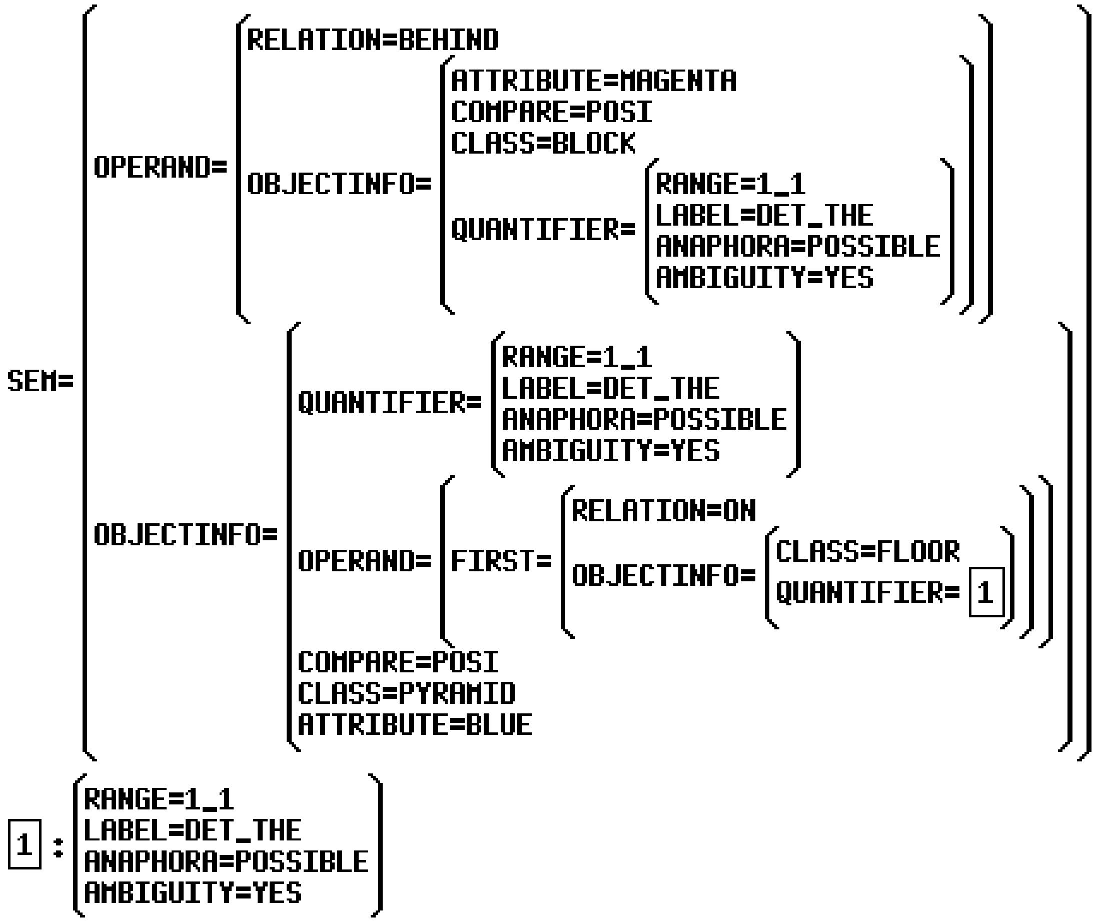


Abbildung: Darstellung einer Feature-Term-Struktur für den Satz:
"The blue pyramid on the floor is behind the purple block."

Nach dem Parserdurchlauf kann man nun mit dem Tool "feat2gif" graphische Darstellungen der Strukturen als GIF-Bilder erstellen, wobei man jeweils angeben muß, welches Feature der obersten Ebene dargestellt werden soll. Dies ist eine speziell für unser System abgestimmte Lösung, da bei unseren Strukturen auf der obersten Ebene zwischen *SYN*, *SEM* und *ERR* unterschieden wird und genau diese Informationen einzeln von Bedeutung für das System sind.

Ein weiteres Feature dieses Tools ist die Eigenschaft, die Strukturen so weit wie möglich zu reduzieren. So werden Feature ohne Wert entfernt. Weiter werden Feature, die als Wert eine Struktur enthalten, die leer ist, entfernt. Dieser Vorgang ist rekursiv geschachtelt. Weiter wurden "unwichtige" Informationen aus den Strukturen entfernt. Dies ist bei uns z.B. die Information *COUNT=0;* im *ERR* Feature. Durch das Herausnehmen dieser Information wurde es mögliche die Fehlerstruktur von korrekten Parses auf die leere Struktur zusammenzukürzen, was das Betrachten der Strukturen ungemein erleichtert.

Sind die Strukturen zu stark geschachtelt und würde die Grafik somit eine zu große Breite erhalten, werden Teilstrukturen über eine Referenz ausgelagert (siehe Beispiel).

Anhang A

Im Verlauf des Projekts entstandene Papiere

A.1 Fehler – Transfer – Interferenz *Alexander Krumeich, Georg Rehm* 13.11.1995

A.1.1 Der "Fehler"-Begriff im CALL-Kontext

"Sprache" wird allgemeinhin in zwei Bereiche unterteilt:

- **Sprachsystem** - definiert die interne Ordnung sprachlicher Elemente und ihre Zusammenhänge auf allen Beschreibungsebenen. Ein Sprachsystem wird in der Linguistik als ein Regelwerk aufgefasst, welches die Menge aller produzierbaren Äußerungen einer Sprache beschreibt.

- **Sprachnorm** - beschränkt die Menge aller möglichen Sprachproduktionen auf der Grundlage bestimmter situativer, funktionaler und thematischer Aspekte.

Es erscheint äußerst problematisch, eine Regelhaftigkeit in der Sprachnorm zu entdecken, da die Bewertung einer sprachlichen Äußerung im Sinne der Norm ein rein subjektiver Prozeß ist. Können wir im Kontext eines CALL-Systems Fehler nur mit Hilfe des regelbehafteten Sprachsystems entdecken und analysieren? (Die Definition des Fehlerbegriffs wird hierbei durch das Komplement zur Menge aller korrekten sprachlichen Äußerungen abgedeckt)

Da wir im CALL-Kontext eine Sprache niemals vollständig beschreiben können, müssen wir uns auf gewisse Teilaspekte einer Sprache beschränken (→ situativer Kontext). Unterliegen wir dabei dem Zwang, auch die Fehlererkennung auf eben diesen Teilaspekt zu beschränken, oder können wir die Behandlung von Schülerfehlern "globaler" vornehmen?

Ill-formedness

- *absolute ill-formed* - ein wissensbasiertes Sprachlehrystem bekommt eine fehlerhafte (*absolute ill-formed*) Eingabe und kann diese mit der vorhandenen Grammatik nicht verarbeiten.

- *relative ill-formed* - das System kann eine Eingabe, die syntaktisch sowie semantisch korrekt ist, nicht verarbeiten, da das zugrundeliegende Fragment der Sprache diesen Teil nicht enthält.

A.1.2 Fehlertypen

aus linguistischer Sicht

Orthographische Fehler Diese Fehler betreffen das fehlerhafte Schreiben eines Wortes. Buchstaben werden verdoppelt, weggelassen, vertauscht oder an einer falschen Stelle hinzugefügt.

- bicycle: *bic*i*cle, *bic*c*ycle, *b*y*cicle

Oft ist es schwer zu unterscheiden, ob ein Orthographiefehler oder ein Fehler aus einer anderen Kategorie vorliegt:

- *He sing a song.

Bei diesem Satz könnte der Schüler zum einen das Singular-Präsens "s" vergessen haben (→ Orthographiefehler), es könnte jedoch auch ein syntaktischer Fehler vorliegen (Kongruenz Subjekt/Prädikat).

Obwohl nicht von linguistischer Bedeutung, sollte an dieser Stelle noch der **Tippfehler** erwähnt werden. Er ist der wohl grundlegendste Fehlertyp. Wenn der Schüler das Falsche eintippt, obwohl er das Richtige gemeint hat, handelt es sich dann um einen Orthographiefehler (s.a. *Performanzfehler*)?

Morphologische Fehler Es liegen Fehler bei der Wortbildung (Komposition (1) und Derivation (2)) oder der Flexion (Deklination (3), Konjugation (4) und Komparation (5)) vor. (Desweiteren handelt es sich bei (2) und (5) vermutlich um Interferenzfehler.)

1. shoulder + blade → *shoulder*'s* blade (statt: shoulder-blade)

2. fear → *fear*ly* (statt: fearful)

3. bus → *buses (statt: busses)

4. go → *he gos (statt: he goes)

5. strong → *strongster (statt: strongest)

Lexikalische Fehler Wird ein falsches Wort in einem Satz verwendet, so spricht man von einem lexikalischen Fehler. Der Fehler ist dabei häufig semantischer Art und nur über den Kontext zugänglich.

- He is a *fearful* guy. vs. He is a *fearless* guy.

Syntaktische Fehler Es finden Verletzungen von Kongruenz-, Wortstellungs- und Satzkonstruktionsregeln statt.

- *I give to him it. (Wortstellungsfehler: direktest/indirektes Objekt)
- *He gives. (Satzkonstruktionsfehler: Mißachtung der Selektionsrestriktionen)

Semantische Fehler Hierzu zählen unter anderem Verletzungen von Selektionsrestriktionen von Verben. Liegen in einem Satz semantische Fehler vor, so bedeutet dies nicht gleichzeitig, daß der Satz auch syntaktisch oder orthographisch nicht korrekt ist.

- *The man smokes a car. (Verletzung der Selektionsrestriktionen des Verbes durch das Objekt)

Die momentan noch bestehenden Probleme im Bereich der Fehlerbehandlung macht folgendes Beispiel klar:

- *The man drives his *cat* to the gas station.

Welche Art Fehler liegt vor? Folgende Möglichkeiten stehen zur Auswahl:

1. Tippfehler: das "r" und das "t" liegen auf der Tastatur nebeneinander, so daß der Schüler aus Versehen den falschen Buchstaben eingetippt hat.
2. Orthographischer Fehler
3. Semantischer Fehler

Mit Hilfe welcher Methoden könnte es nun einem CALL-System möglich sein, eine gut begründete Fehlerhypothese über diesen Satz aufzustellen? Angenommen, der Schüler hat die vergangenen Übungen in einer recht kurzen Zeit erfolgreich absolviert, so erscheint es eher unwahrscheinlich, daß ein semantischer oder ein orthographischer Fehler vorliegt (man könnte hier z.B. die bislang vom Schüler gemachten und danach klassifizierten Fehler in einem Benutzermodell sammeln, um aus diesen statistischen Daten eine Wahrscheinlichkeit über den jetzt zu analysierenden Fehler — sofern schon erkannt — zu berechnen). Vielmehr könnte es sich hier um einen Tippfehler handeln. Die Fehlererkennung könnte nach dem Parsen des (orthographisch und syntaktisch korrekten) Satzes mit Hilfe von Selektionsrestriktionen erfolgen ("drive" erfordert als Objekt eine Instanz aus der Kategorie *Fahrzeug*; vorhanden ist "cat"; gibt es im Lexikon ein ähnliches Wort aus der Kategorie *Fahrzeug*? $\rightarrow$ "car"). Daraufhin könnte dem Schüler vom System eine Meldung ausgegeben werden, daß er vermutlich einen Tippfehler begangen hat. Das System darf dabei jedoch nicht den Anschein erwecken, bei mehreren möglichen Erklärungen für einen Fehler immer die korrekte zu erkennen.

Pragmatische Fehler Es liegt ein syntaktisch und semantisch korrekter Satz vor, der jedoch der Formulierung, die ein Muttersprachler in diesem Kontext verwenden würde, widerspricht. Will man pragmatische Fehler in einem CALL-System behandeln, so ist es unbedingt erforderlich, eine sprachliche Norm in das System zu integrieren, die als Meßlatte für pragmatische Fehler fungiert.

Zu überlegen wäre in diesem Zusammenhang noch, welche Altersstufe mit dem CALL-System angesprochen werden soll. Sind Schulkinder, die gerade die ersten Erfahrungen im Englischen machen, unsere Zielgruppe, so können wir unter Umständen auf eine Pragmatikkomponente verzichten, wohingegen eine solche Komponente für ein rein situatives CALL-System (z.B. "Englisch für Manager" mit solchen Übungen wie "Terminabsprache" oder "Briefing") sehr wohl von Bedeutung sein könnte.

Stellen wir uns ein "Universal-CALL-System" mit solch verschiedenen Übungen wie "Vorstellungsgespräch", "Restaurantbesuch" oder "Geburtstagsfeier" vor, so ist die wohl einfachste Möglichkeit, mit Pragmatik ansatzweise umzugehen, die, daß wir bestimmte Phrasen in bestimmten Kontexten (= Übungen) als zulässig erlauben oder nicht. Natürlich bedeutet dies, daß wir nur rein antizipatorisch mit Pragmatik (auf einem sehr niedrigen Level) umgehen könnten. Ein nur teilweise oder gar nicht antizipierendes System wäre hier sicherlich wünschenswerter. Das würde dann widerum bedeuten, daß wir eine (möglichst umfassende) Repräsentation der *sprachlichen Norm* (s.o.) in das System integrieren müssen.

Fehlertypen aus Sicht der Form

Die Form eines Fehlers kann zur Erklärung und Korrektur herangezogen werden. Folgende Bereiche lassen sich dabei unterscheiden:

- **Vertauschungen** - *you *it got.* statt: you got it. *thou*h*gt statt: thought
- **Auslassungen** - *the king France statt: the king of France.
- **Einfügungen** - *London's *in* greatest warehouse statt: London's greatest warehouse

Vertauschungen könnten unter Umständen durch Operationen auf dem Lexikon eines CALL-Systems identifiziert werden (das System bekommt als Teil des Inputs das Wort **thouhgt* und könnte es nach Durchsuchen und Permutieren der in Frage kommenden Lexeme korrekt als *thought* identifizieren. Im nächsten Schritt müßte das System versuchen, eine Hypothese darüber aufzustellen, ob der Schüler *thought* als Verb oder als Substantiv benutzen wollte. Dazu müsste der Inputsatz (falls überhaupt schon geschehen) erneut — unter Umständen sogar mehrmals — geparsed werden.

Auslassungen könnte man mittels Selektionsrestriktionen erkennen.

Einfügungen könnte das System nur dann verarbeiten, wenn gezielt Teile des Inputs eliminiert werden, wonach das System einen erneuten Parseversuch durchführt. Es erscheint unklar, nach welchen Kriterien diese Eliminationen vollzogen werden könnten.

Fehlertypen aus Sicht der Art des Fehlers

Will man noch den Zustand des Schülers (Müdigkeit, Unkonzentriertheit, ...) bei der Klassifizierung von Fehlern berücksichtigen, so ergeben sich neue Unterscheidungsmöglichkeiten:

- Kompetenzfehler — der Schüler hat wider besseren Wissens eine fehlerhafte Konstruktion oder ein falsches Wort benutzt und somit einen Fehler verursacht.
- Performanzfehler — der Schüler war unkonzentriert oder unter Zeitdruck als er den Fehler machte, ist sich aber der korrekten Verwendungsweise oder des korrekten Wortes bewußt.

Um Fehler nach diesen beiden Gesichtspunkten zu klassifizieren, muß man unter anderem berücksichtigen, wie lange der Schüler schon mit dem System arbeitet (Abbau der Konzentrationsfähigkeit), wie die Übung strukturiert ist (→ möglicher Zeitdruck) und welche Fehler der Schüler in den bisherigen Übungen gemacht hat (→ Lernermodell).

Diese Unterscheidung in Kompetenz- und Performanzfehler ist vor allem für orthografische Fehler interessant, da das Eingeben von Sätzen in der zu erlernenden Sprache mittels der Tastatur für den Schüler u.U. neu ist und somit fehlerprovozierend wirkt (→ Flüchtigkeitsfehler). Durch die beiden neuen Begriffe könnten nun Aussagen zu Wiederholungsfehlern und auch allgemeinen Schwächen des Schülers gemacht werden.

Krüger-Thielmann fügt an, daß es bei gewissen Fehlern sehr schwer ist, zwischen Kompetenz und Performanz zu unterscheiden.

- **environnment* - hat der Schüler die "n"-Taste einmal zu oft betätigt oder glaubt er, daß man *environment* mit zwei "n" schreibt?

Fehlertypen aus Sicht der Ursache

Ein auftretender Fehler kann die verschiedensten Ursachen haben. Der Schüler könnte z.B. an einer Lernschwäche (Legasthenie) leiden. Findet ein Muttersprach-, Zweitsprach- oder Drittspracherwerb statt? Auch das persönliche Profil des Schülers kann hier eine Rolle spielen. Häufig auftretende Fehler sind die sog. Interferenzfehler (s.u.).

A.1.3 Interferenzfehler

Begriffe

Transfer - bezeichnet die Übertragung von Sprachstrukturen von einer Sprache (auch: *Interferenzbasis*; im Regelfall die jeweilige Muttersprache) in eine andere Sprache (auch: *Zielsprache*).

- **Positiver Transfer** - Sprachtransfer führt zu einer korrekten sprachlichen Äußerung in der Zielsprache
- **Negativer Transfer (Interferenz)** - die gebildeten Strukturen in der Zielsprache sind fehlerhaft. Interferenzfehler können auf sämtlichen linguistischen Beschreibungsebenen stattfinden.

– **Intralinguale Interferenz** - Fehler, die aus bereits gelernten Strukturen der Zielsprache entstehen

– **Interlinguale Interferenz** - Strukturen der Muttersprache, bzw. einer weiteren, bereits gelernten Fremdsprache, sind verantwortlich für fehlerhafte Konstruktionen in der Zielsprache. Ist von *Interferenz* die Sprache, so meint man im Allgemeinen interlinguale Interferenz.

Fehlerhäufigkeit von Interferenzen in ...

Jörger ('87) stellte in einem Versuch folgende Fehlerhäufigkeiten von Interferenzen fest:

...der Orthographie: 30,17 %	...der Lexik: 30,84 %
...der Morphologie: 26,81 %	...der Syntax: 4,08 %

Der Grund, weshalb orthographische und lexikalische Interferenzen so häufig auftreten liege darin begründet, daß Vokabeln wesentlich schwerer zu erlernen seien, als z.B. grammatikalische Strukturen. Da Vokabeln in einer sehr großen Menge auf den Schüler einprasseln, würde er diese auch eher verwechseln.

Warum findet Interferenz statt?

Strukturen aus Grund- und Zielsprache sind einander ähnlich, was beim Schüler Fehler provoziert (→ *partielle Identität*). Hierbei steigen die Lernschwierigkeiten mit zunehmender Ähnlichkeit.

Ein Beispiel für partielle Identität ist das Divergenzproblem (auch: Problem der multiplen Diskrimination). Zwischen verschiedenen Lesarten eines Begriffs in Grund- und Zielsprache herrscht eine mangelnde Übereinstimmung. Im Deutschen haben wir z.B. die Präposition "in" wohingegen im Englischen so verschiedene Begriffe wie "into", "at", "in" oder auch "to" zur Verfügung stehen. Die erste Lesart, die ein Schüler lernt ist wohl "in" (dtsch.) = "in" (engl.). Verwendet der Schüler nun in einem englischen Satz "in" anstatt "into", so könnte er sich der weiteren Vokabelentsprechungen noch nicht bewußt sein.

Interferenzfehlerklassen

Orthographische Interferenzen Orthographische Interferenzen sind zurückzuführen auf partielle Identitäten in der Typographie oder der Phonologie zweier Worte aus Grund- und Zielsprache. Die Bedeutungsunterschiede zwischen den beiden Worten können u.U. erheblich sein.

- Adresse → *a*d*ress (statt: address)
- Sonne → *sunn* (statt: sun)
- Sommer → *s*o*mmer (statt: summer)

Morphologische Interferenzen Es findet eine Übertragung von Wortbildungs- oder Flexionsregeln von der Interferenzbasis in die Zielsprache statt. Wie bereits erwähnt ist hier die Abgrenzung zu orthographischen Fehlern oftmals problematisch. Morphologische Fehler sind oft Kompetenzfehler, wohingegen (einbuchstabige) Orthographiefehler auch Performanzfehler (= Tippfehler) sein können.

- Peters Geburtstag → *Peter*s* birthday (statt: Peter's birthday)
- Er fährt nach Frankreich. → He *goe*t* to France. (statt: He goes to France.)

Lexikalische Interferenzen Diese Fehler ergeben sich durch den Gebrauch eines falschen Wortschatzes in der Zielsprache und können oft nur mit Hilfe des Kontextes erkannt werden. Zu diesem Fehlertyp zählen auch "false friends" (bekommen → to become). Die falschen Freunde entstanden im Laufe der Jahrhunderte durch Sprachwandel. Ihnen ist die äußere Form (typographisch oder phonologisch) gemeinsam, wobei sie sich aber in der Bedeutung unterscheiden. Diese partielle strukturelle Identität wird vom Schüler fälschlicherweise auf die Semantik übertragen.

- Lexikal. Interferenzen bei wörtlichen Übersetzungen von idiomatischen Ausdrücken
  - It's raining cats and dogs. → Es regnet *Katzen und Hunde. (statt: Es regnet Bindfäden.)
  - Ich weiß nicht so recht. → *I don't know so exactly. (statt: I'm not sure at all.)
- "false friends"
  - Gymnasium → *gymnasium (statt: high school)
  - aktuell → *actual (statt: topical)
  - etwas überhören → *to overhear sth. (statt: to miss hearing sth.)
  - bekommen → *to become (statt: to get, etc.)
  - handeln → *to handle sth. (statt: to act)[1]
- "false friends" mit ähnlichen Bedeutungen
  - Salat → *salad (gemeint ist "frischer Salat") (statt: lettuce (Kopfsalat))
  - Ambulanz → *ambulance[2] (gemeint ist "Behandlungsstelle für Patienten") (statt: outpatient's department)

[1] hier haben wir neben dem Bedeutungsunterschied der beiden Verben auch einen syntaktischen Unterschied: "handeln" ist ein intransitives, "to handle" ein transitives Verb

[2] lediglich in der Lesart "Krankenfahrzeug" besteht zwischen 'Ambulanz' und 'ambulance' noch eine semantische Übereinstimmung

Syntaktische Interferenzen Interferenzfehler auf syntaktischer Ebene lassen sich durch Vertauschungen, Auslassungen oder Einfügungen von verschiedenen Satzgliedern charakterisieren. Der Schüler benutzt also z.B. einen falschen Satzbau, falsche Präpositionen oder verwechselt Adjektiv und Adverb. Syntaktische Interferenzen lassen sich zum Großteil strukturell feststellen.

- Sie hat eine Pause gemacht. → *She has a break taken. (statt: She has taken a break.)
- Hast Du das Auto gesehen? → *Have you the car seen? (statt: Have you seen the car?)
- Kennst Du diesen Mann? → *Know you this man? (statt: Do you know this man?)

Failed der Parsevorgang bei einem fehlerhaften Input, so könnte man versuchen, die Struktur des Satzes mit einer Grammatik zu parsen, die der Muttersprache des Lerners entspricht. So könnte man u.U. syntaktische Interferenzen erkennen und entsprechend darauf reagieren. In dem CALL-System PROMISE wurden einzelne Lexikoneinträge mit Fehlerregeln annotiert, um bestimmte Schülerfehler zu antizipieren. Diese Methode funktioniert dadurch nur sehr eingeschränkt. Die Implementierung einer zweiten Grammatik, die dann indirekt als Fehlergrammtik fungiert, ist sicherlich aufwendiger, erkennt aber syntaktische Interferenzen in einem nicht so eingeschränkten Kontext und kann in der Fehlerbehandlung flexibler reagieren. Mit einer solchen Methode beweget man sich aus dem Bereich der Faktenantizipation hinaus in den Bereich der Regelantizipation.

Semantische Interfernzen Semantische Interferenzen treten in Sätzen auf, die der Form nach syntaktisch und lexikalisch korrekt, jedoch z.B. im Sinnzusammenhang des Gesamtkontextes falsch sind (2), (3). Oftmals handelt es sich hierbei um Verletzungen der Selektionsbeschränkungen des Verbes (1). Wie bei allen Fehlertypen ist auch hier der Übergang zu anderen Fehlertypen fließend.

1. Das Auto fuhr auf den Hund zu → *The car was driving towards the dog. (statt: The car was going towards the dog.)
2. Der Hund rannte auf die Straße. → *The dog ran on the street. (statt: The dog ran onto the street.)
3. Hier ist Mrs. Hamilton. → *Here is Mrs. Hamilton.[3] (statt: This is Mrs. Hamilton.)

A.1.4 Empfehlung für das Projekt

Inwieweit unterstützt CoALA die intelligente Behandlung von Fehlern?

- Integration eines Lernermodells in CoALA
- Fehlerdiagnose und Bewertung von Fehlerhypothesen durch PROLOG-artige Auswertungsregeln (innerhalb des "Evaluators")

[3] fließender Übergang zu lexikalischen Interferenzen (s.o.)

– Klassifikation von Fehlerhypothesen z.B. *Systematischer Fehler*, *Wortform*, *Verwechslung Plural-s / Genitiv-s*
– Auswertungsregeln stellen einen Zusammenhang zwischen Lernzielen und Fehlerhypothesen her
– Filtermechanismen setzen bei Vorliegen konkurriender Fehlerhypothesen Prioritäten basierend auf den zugrundeliegenden Lernzielen und den Inhalten des Lernerprofils

Welche Fehlerklassen wollen wir behandeln?

(Dieser Abschnitt wird zu gegebener Zeit nachgereicht)

A.1.5 Literatur

- Bauer, Pascal: Fehlerbehandlung in CALL-Systemen, Magisterarbeit. Universität Osnabrück 1995.
- Krüger-Thielmann, Karin: Wissensbasierte Sprachlehrsysteme, Gunter Narr Verlag Tübingen 1992
- Jörger, Uwe: Can I become a beefsteak?, L.A.U.D. Duisburg 1987
- Unsöld, Robert: Wie soll das ICALL-System aussehen? Universität Osnabrück (erstellt im Rahmen eines Seminars) 1995
- Folien der IBM zur Präsentation des CoALA-Konzepts auf dem ILL-Workshop (Mai 1995) [4]

A.2 Unvollständige Auflistung von sinnvollen Eingaben in der Blocks World *Henrik Dittmann* 27.1.1996

A.2.1 Aussagen

Existenz: *There is a block. There are many green blocks there.*

Eigenschaft: *The yellow cube is quite large. There is not a single object which is light brown.*

Relation: *A pyramid is near to a blue ball. The floor is beneath all the squares.*

Anordnung: *All the cubes form a circle.*

[4] es bleibt zu prüfen, ob und wenn ja, inwieweit Teile dieses Konzeptes bereits revidiert wurden

Vergleich: *There are more green cylinders than yellow ones. The purple ball is the largest.*

Tautologie: *I can see all the blocks.*

A.2.2 Aktionen

Bewegung: *Move the green box to the left. Next I push the block a bit. Turn the turquoise cone.*

Bewegung relativ zu einem Objekt: *I pull the box away from the white pyramids. I wish to put the pyramid on top of the leftmost ball.*

Aktionen eines Greifarmes: *Grab the pink cylinder. I drop all the objects I am holding.*

Anordnung: *Now I want to align all the cones. Stack all the boxes.*

A.2.3 Fragen

Versteckte Aussage: *Is the blue cube on top of the orange ball?*

Versteckte Aktion: *Can I/you place the small cubes next to the biggest cone?*

Quantität: *How many brown pyramids are there standing in the back?*

Qualität: *What colour is the object next to the pillar? How is the thing in front of the huge pyramid called?*

Kausalität: *Why can't I push the cone through the ball?*

A.3 Vom Semf-System zu behandelnde Fehlertypen *Klaus Dalinghaus, Henrik Dittmann, Alexander Krumeich* 14.3.1996

Diese Beispieleingaben sollen vom Semf-System verarbeitet und entweder als richtig erkannt oder mit einer Fehlerbeschreibung belegt werden können. Sie stellen eine Minimalanforderung an die Fähigkeiten des Systems dar. Darüberhinausgehend ist vor allem ein Augenmerk auf Sätze zu legen, die mehrere der unten genannten Fehlertypen enthalten, wobei diese abhängig oder unabhängig voneinander auftreten können.

Zu lesen nur in Verbindung mit der Semfwelt[5].

[5]http://www.cl-ki.uni-osnabrueck.de/local/projekt/papiere/welt.ps.gz

A.3.1 Bei der syntaktischen Verarbeitung zu behandelnde Fälle

- (1) *Some objects are standing in front of the largest blue block.*
  (2) *The ball is between the green and the yellow blocks.*
  (korrekte Sätze)

Kongruenzfehler

WERDEN BEIM PARSING ERKANNT

- (3) **A balls is on the floor.*
  (keine Übereinstimmung im Numerus zwischen *Det* und *N*)

- (4) **The block are standing on the floor.*
  (keine Übereinstimmung im Numerus zwischen *NP* und *V*)

Wortstellungsfehler

WERDEN MIT DER FEHLERGRAMMATIK GEPARST

- (5) **There is standing a block on the floor.*
  (Subjekts-*NP* an der falschen Stelle)

- (6) **The block is somewhere standing.*
  (*Adv* an der falschen Stelle)

- (7) **The block on the floor standing is high.*
  (*PP* an der falschen Stelle)

Satzkonstruktionsfehler

WERDEN MIT DER FEHLERGRAMMATIK GEPARST

- (8) **The block is stand on the floor.*
  (*progressive form* ohne *-ing*)

- (9) **The block is stands on the floor.*
  (überflüssiges *Aux* - vgl. auch (8))

- (10) **The block is next to ball.*
  (fehlender *Det*)

- (11) **The red object is on top of the green.*
  (fehlendes *N* - vorbehaltlich Dennis' Zustimmung ;-))

- (12) **The block on the floor.*
  (fehlendes *V*/fehlende *VP*)

- (13) **The block is on the right at the ball.*
  (14) **The blue block is behind of the ball.*
  (Kombination von Präpositionen - ERKENNUNG DURCH FEHLERGRAMMATIK ODER PARSING (LEXIKON!))

Selektionsrestriktionsverletzungen

WERDEN BEIM PARSING ERKANNT (LEXIKON!)

- (15) **The yellow block touches to the ball.*
  (Subkategorisierung bei Verben)

- (16) **The ball stands on the floor.*
  (semantische Features im Lexikon verletzt)

- (17) **The ball lies between the cube.*
  (Komplementforderungen der Präposition verletzt)

- (18) ***The large block is between the blue block and the green cube*[6].
  (19) ***The pyramid is between the green cubes.*
  (entspricht nicht dem vorgegebenen Kontext - BEHANDLUNG WIE FÄLLE (20) BIS (37))

A.3.2 Bei der semantischen Verarbeitung zu behandelnde Fälle

Quantifikation und Negation

(20) UND (21) WERDEN GGF. BEI DER SEMANTIKKONSTRUKTION ERKANNT, DIE ANDEREN, SOWEIT BEHANDELBAR, VON DER INFERENZMASCHINE

- (20) ***Blue objects are blue.*
  (21) ***Green blocks are not yellow.*
  (allgemeine Aussagen ohne Bezug zur Welt - ALS KONTEXTUELLE FEHLER ZU BEHANDELN)

- (22) *No cube is on the ball.*
  (23) *Some objects are blue.*
  (24) *Most objects are blocks.*
  (25) *All the blocks are on the floor*
  (26) *The blue objects are blue.*
  (27) *The green cube is not red.*
  (28) *There is only one green cube.*
  (29) *The red pyramid is not on the yellow cube.*
  (faktisch korrekte Sätze)

[6] „**“ denotiere einen zwar grammatischen, aber im vorgegebenen Kontext nicht sinnvollen Satz.

- (30) ***The green cube is red.*
  (faktisch falscher Satz - KANN MÖGLICHERWEISE SCHON BEI DER SEMANTIKKONSTRUKTION ERKANNT WERDEN)

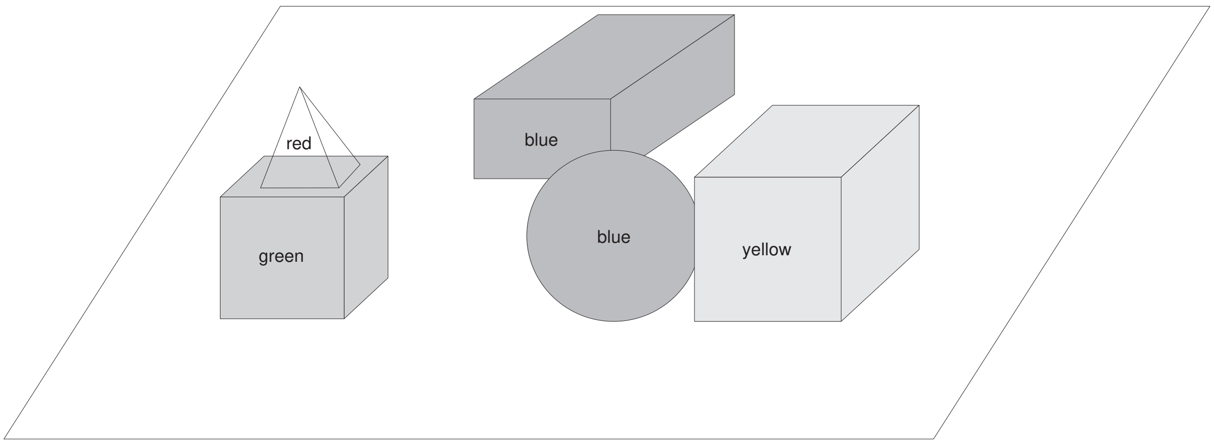
Im Kontext falsche Sätze

WERDEN VON DER INFERENZMASCHINE ERKANNT

„Wahrheit" gestaltet sich als ein Kontinuum, das von *falsch* über individuell unterschiedliche Konzepte von Raum („Für mich ist das jetzt aber nicht mehr links..") bis zu *wahr* reicht. Interferenzmaschine und Fehlererklärung müssen mit dieser Unschärfe umgehen können.

- (31) *Next to the yellow object is a blue ball.*
  (32) *In front of the largest block are the green cube and the pyramid.*
  (korrekte Sätze)
- (33) ***The red ball is on top of the green cube.*
  (falscher Objektbegriff)
- (34) ***Next to the yellow block is a red ball.*
  (falsches Attribut)
- (35) ***The blue ball touches the blue block.*
  (36) ***The ball lies before the biggest thing.*
  (falsche Relation)
- (37) ***All objects are on the floor.*
  (falsche Quantifizierung)

A.4 Die Beispielwelt *Alexander Krumeich* 7.3.1996



A.5 Zur Koordination von Nominalphrasen *Alexander Krumeich* 10.7.1996

Bei der Behandlung von koordinierten Nominalphrasen stellt sich zuerst die Frage, wie die Phrasenstruktur der NP aussehen soll. Der Einfachheit halber habe ich mich für die Struktur NP $\rightarrow$ NP CONJ NP entschieden. Es bleibt zu klären, ob die Assoziativität des resultierenden Baums die Äußerung adäquat wiedergibt, doch das ist mir für den Augenblick egal.

Die zweite Frage, die sich aufdrängt, lautet: Was ist der Kopf einer koordinierten NP? Um hier keine Entscheidung fällen zu müssen, habe ich mich entschieden, ein Subfeature `COMPOUND` einzuführen, das wiederum genau zwei Subfeatures `OBJ1` und `OBJ2` enthält. In diese beiden Features werden die `OBJECTINFO`-Features der koordinierten NPn unifiziert. Die nicht-kompositionelle Semantik wird hier die Aufgabe haben, diese Struktur aufzulösen.

Problematisch wird die Behandlung koordinierter NPn, wenn eine NP „kopflos" ist, wie z.B. im Satz

The red and the green cube are on the floor. (A.1)

Bei der Implementierung der Regeln bin ich davon ausgegangen, daß eine solche Konstruktion nur gültig ist, wenn bei der ersten NP der Kopf fehlt. Einen Satz wie

**The red cube and the green are on the floor* (A.2)

halte ich für ungrammatisch, lasse mich aber gerne belehren... Für Fall A.1 habe ich angenommen, daß die erste NP folgendermaßen zustande kommt: $\bar{\mathrm{N}} \rightarrow \mathrm{A}$, wobei $\bar{\mathrm{N}}$ das syntaktische Feature `HEADLESS` bekommt und gleichzeitig die `CLASS` auf den Dummy-Wert `NO_HEAD` gesetzt wird.

Damit die Grammatik nun nicht hemmungslos übergeneriert[7], wird bei der Konstruktion des Satzes verlangt, daß die Subjekt-NP nicht `HEADLESS` ist.

Neben der Regel für die Koordination zweier NPn mit jeweils einem Kopf gibt es eine zweite Regel, die abprüft, ob der ersten NP der Kopf fehlt. Wenn dies der Fall ist, wird ebenfalls eine `COMPOUND`-Struktur als `OBJECTINFO` angenommen. Der Dummy-Wert im `CLASS`-Feature der ersten NP wird dann durch den Wert des `CLASS`-Features der zweiten, vollständigen, NP ersetzt, so daß nach Anwendung der Regel nicht mehr auszumachen ist, ob die NP vorher „kopflos" war oder nicht.

Noch nicht geklärt sind bislang folgende Fragen:

- Was geschieht mit dreifach koordinierten NPn der Form *The red and the green and the blue cube*? Hierbei sind beide Konstituenten der ersten NP kopflos.
- Wie verhindert man beim Satz *The red and the green cube are blue* eine Analyse folgender Art: (Analyse von blue als NBar statt als Adjektiv, Baum zeichnen)
- Wir sollten nur einfarbige Objekte zulassen, sonst gibt es Ärger mit Sätzen wie *The red and green cubes are on the floor.* Dieser Satz besitzt mindestens zwei Lesarten.

[7] z.B. im Satz **The green is a red.*

A.6 Funktionalität der syntaktischen Fehlerbewertung *Henrik Dittmann, Michael Huber* 12.7.1996

Da uns zwischenzeitlich die Ergebnisse unseres Treffens abhanden gekommen sind, ist dies nur eine vorläufige Version ;-(.

A.6.1 Funktionen, die über Regelanwendung entscheiden

- realisiert als Funktionsaufrufe im Kopf von Fehlerregeln

1. Fehlerregeln werden nicht mehr angewandt, wenn die aktuelle Interpretation bereits zuviele Korrekturen enthält
   → diese Alternative wird vom Parser nicht mehr weiterverfolgt.

A.6.2 Funktionen, die Werte für das Error-Feature berechnen

- realisiert als funktionale Ausdrücke im Rumpf von Fehlerregeln

1. Die Fehlerhypothesen werden bei Anwendung einer Fehlerregel mit einer Wahrscheinlichkeit versehen, die sich zunächst aus der Wahrscheinlichkeit der Fehlerregelanwendung ergibt, später ist auch eine Modifikation dieses Wertes denkbar, abhängig von z.B. den bereits angewendeten Fehlerregeln oder übungsabhängigen Parametern (Lernziele).

2. Zusätzlich sind Funktionen notwendig, um die Fehlerstelle im Satz und ggf. Korrekturen und detaillierte Fehlerbeschreibungen zu errechnen.

A.6.3 Funktionen nach Beendigung der Satzanalyse

- also solche, die von einer Komponente ausgeführt werden, die nach Abschluß der Verarbeitung durch den Parser und vor Einsetzen der Analyse durch die kompositionelle Semantik aufgerufen wird

1. Die Error-Features einer Parse-Alternative werden unter Vergabe einer gemeinsamen ID in einer temporären Datei abgelegt. Für die weitere semantische Verarbeitung dagegen ist nur das Sem-Feature notwendig. Bei der Verwaltung der IDs ist besonders zu beachten, daß bei der Analyse durch die nicht-kompositionelle Semantik weitere Alternativen (mit neuer ID) entstehen können.

2. Falls in allen generierten Alternativen hinreichend schwere Fehler auftreten, wird die semantische Analyse gar nicht erst aufgerufen, sondern direkt eine Mitteilung ausgegeben.

3. Sofern eine syntaktische Korrektur möglich ist, muß auf Informationen aus der Chart zugegriffen werden, weshalb die Korrektur an dieser Stelle stattfinden sollte. Aufgrund des komplexen Aufbaus der Chart und der Probleme bei Kategorieübergreifenden Fehlern ist diese Aufgabe jedoch zurückzustellen.

Für die korrekte Ermittlung von Quantoren für NPn, deren Numerus als falsch erkannt wurde, benötigen wir in der Grammatik ein SEM→QUANTIFIER→QTYPE-Feature, das uns erlaubt, Quantoren, die in verschiedenen Numeri existieren, von solchen zu unterscheiden, die eindeutig den Numerus der NP festlegen (one, two, some, most, every/all (?)).

A.7 Beispielanfragen an den Inferenzserver *Andreas Rother* 17.9.1996

A.7.1 Anfragen an die Inferenzmaschine

Die folgenden Testsätze passieren die Inferenzmaschine und liefern bis auf Satz 1 als Resultat der Anfrage `ok`. Satz 1 liefert `fail`. Die dazugehörigen Welten befinden sich in der neuesten Version des Inferenzpakets und können mit `load Scene 1|2|3` geladen und dann mit `display #handle` angeschaut werden.

Die Syntax der Anfragen an die Inferenzmaschine ist sehr restriktiv. Groß- und Kleinschreibung ist zu beachten, ebenso dürfen keine überzähligen Leerzeichen vorkommen.

A.7.2 Testsätze

Einige Attribute und Entitäten sind im Vergleich zur vorigen Liste der Testsätze geändert worden, um die Welt einfacher zu gestalten. Die Fehlerklassen der Beispielsätze sind jedoch gleich geblieben.

```
1)  The yellow block is bigger than the red pyramid on top of
    the cyan block.

    load Scene 1
    infer 0 (Block 1 1 e color=yellow|bigger|(Pyramid 1 1 e
    color=red|on|Block 1 1 e color=cyan))

2)  The pink ball and the yellow cube are in front of the
    blue ball.

    load Scene 1
    infer 0 (Ball 1 1 e color=pink+Cube 1 1 e color=yellow
    |front|Ball 1 1 e color=blue)
```

```
3)  The cyan and the yellow block are standing in front of a
    block.

    load Scene 1
    infer 0 (Block 1 1 e color=cyan+Block 1 1 e color=yellow
    |front|Block 1 %100 e)

4)  Green blocks are green.
4') There is a green block.

    load Scene 1
    infer 0 (Block 1 %100 e color=green|exist)

5)  There is block.

    load Scene 1
    infer 0 (Block 1 %100 e|exist)

6)  All pyramids are on a block.

    load Scene 2
    load Scene 3
    infer 8 (Pyramid %100 %100 g|on|Block 1 %100 e)
    infer 0 (Pyramid %100 %100 g|on|Block 1 %100 e)
    infer 8 (Pyramid %100 %100 e|on|Block 1 %100 e)
    infer 0 (Pyramid %100 %100 e|on|Block 1 %100 e)

7)  The on the floor standing blue pyramid is behind the cube.

    load Scene 1
    infer 0 ((Pyramid 1 1 e color=blue|on|Floor 1 1 e)|behind|
    Cube 1 1 e)

8)  No object touches between the pyramids.

    load Scene 1
    infer 0 (Entity 0 0 e|between|Pyramid 1 %100 g)
    infer 0 (Entity 0 0 e|touch|Pyramid 1 %100 g)
```

Anhang B

Autorenverzeichnis

1	Einleitung			A. Krumeich G. Rehm
2	Architektur			G. Bongardt
3	Master-Komponente			W. Teiken
4	Der Parser			W. Teiken
5	Grammatikformalismus			W. Teiken
6	Grammatik			
		Syntax		H. Dittmann
		Semantik	Motivation	K. Dalinghaus
			Funktionalität	H. Dittmann
7	Lexikon			A. Krumeich
8	Nichtkomp. Semantik			G. Rehm
9	Inferenzmaschine			A. Rother
10	Weltrepräsentation			C. Peylo
11	Fehlerbehandlung			
		Allgemeines		H. Dittmann
		Synt. Fehlerbehandlung		M. Huber
		Sem. Fehlerbehandlung		K. Dalinghaus
		Fehlerbewertung		H. Dittmann
12	Übung			G. Bongardt
13	Benutzeroberfläche			
		WWW-Oberfläche		G. Rehm
		Visualisierung		K. Dalinghaus