Problems of Generating German from First-Order Formulae in Automatic Translation from English to German

B.A. Thesis
Seminar für Sprachwissenschaft
Universität Tübingen

Course: Computational Semantics
Instructor: PD Dr. Frank Richter

Kilian Evang
Kyffhäuserstraße 86a
42115 Wuppertal

Hiermit versichere ich, dass ich die vorgelegte Arbeit selbstständig und nur mit den angegebenen Quellen und Hilfsmitteln einschließlich des WWW und anderer elektronischer Quellen angefertigt habe. Alle Stellen der Arbeit, die ich anderen Werken dem Wortlaut oder dem Sinne nach entnommen habe, sind kenntlich gemacht.

(Kilian Evang)

Contents

1 Introduction

Computational Semantics (CS) is a discipline with the aim "to find techniques for automatically constructing semantic representations for expressions of human language, representations that can be used to perform inference" (Blackburn and Bos, 2003, p. 1). Since semantic representations are representations of the propositional content of natural language expressions, they suggest themselves for another task, namely to automatically generate expressions of a different natural language that convey the same propositional content. This would bring the semantic representation language (SRL) used close to being an interlingua in the context of Machine Translation (MT).

The purpose of the present thesis is to make use of an SRL as an interlingua in order to build a small MT system from English to German. To this end, I have developed a Natural Language Generation (NLG) component for a fragment of German to extend an existing CS system, which is already capable of assigning semantic representations to expressions of a fragment of English.

In theory, an interlingua is a formal language capable of representing linguistic content independently of the particularities of any natural language, thus serving as an intermediate format for translating from any source language for which a suitable analysis component can be made into any target language for which a suitable generation component can be made. In practice, the design of these components is not the only fundamental problem in interlingua MT. Finding a sufficiently expressive and unambiguous interlingua alone necessitates compromises concerning the range of source and target languages, translation quality, and the range of syntactic constructions and vocabulary covered (Hutchins and Somers, 1992, p. 118–124).

Rather than devising an interlingua that meets the requirements of a particular limited translation task, this thesis puts the cart before the horse in opting for a particular SRL as interlingua and trying to find out how far one can get with that in translating sentences automatically. The CS system chosen to be extended is BB1, the Prolog software accompanying the textbook by Blackburn and Bos (2005). The SRL it uses is a language of first-order logic, generated from English sentences using a technique based on Montague semantics (Blackburn and Bos, 2005, p. xii). The remaining task for the NLG component is thus to generate German sentences from first-order semantic representations. No other input is used in the NLG component, in accordance with the modular architecture of interlingua MT systems.

Existing approaches to NLG from first-order formulae, sketched in section 2, point in the direction of some fundamental difficulties that the choice of a logical language as interlingua might create when the system is extended to larger fragments of German or to other languages. Despite such problems, it seems worthwhile to start a new attempt at generating sentences from first-order SRLs in the context of CS. First, the increasing availability of CS applications presently makes it more convenient than ever to experiment with semantic representations, e.g. to derive them from text, perform inference with them or, as in the present case, try MT with them, where BB1 can serve as a readily available analysis component.

Secondly, there certainly is potential for fruitful contact between Computational Semantics on one side and Machine Translation and Natural Language Generation on the other side. For example, a part of BB1 not used in this thesis is Curt, an interactive application used to demonstrate the power of semantic inferencing. One can envision to develop Curt into a real dialog system with near-natural linguistic exchange between human and computer. For this purpose, NLG from the semantic representations used internally would be indispensable. As another example, the inferencing capabilities of CS systems could be used together with world knowledge and situational knowledge for disambiguation in MT.

The structure of the thesis is as follows: Section 2 sketches the development of the most important strand of research in NLG from first-order formulae, shows implications of the choice of SRL for generation, details the SRL chosen in this system and discusses possible alternatives within the realm of first-order languages and close variants. Section 3 proposes a pragmatic solution for generating German from the SRL at hand and describes its architecture and implementation in detail, giving an account of the treatment of different logical and syntactic constructions involved. An evaluation of the system is given in section 4, discussing its place in the MT landscape, fundamental problems, and possible improvements. Section 5 concludes with an assessment of what has been achieved and shown, and what directions seem most promising to explore from there in order to create a useful link between Computational Semantics and Machine Translation.

2 Natural Language Generation from First-Order Formulae

2.1 What to Generate from

The question of what to generate from has been called the "most vexing" (McDonald, 1993, p. 191) one in NLG. The answer given so far for this thesis is "formulae of first-order logic", but this answer is not very specific. Before work can begin on a natural language generator, it needs to be clarified *what* the formulae it is going to be fed with encode and *how* they encode it. In other words, the language of first-order logic to be used as an SRL (a first-order SRL for short) needs to be specified. There are different possible answers, and they can have dramatic consequences for the design of the generator.

First, *what* should the input to the generator in an MT system encode? An obvious answer would be "meaning", but Phillips (1993, p. 219) rejects this term because of its vagueness. In the introduction to this thesis, I used the term "propositional content" because pragmatic or stylistic content is not represented in the SRL here, although such content could be considered part of the meaning of natural language expressions. This seems to be compatible with Phillips's more concrete answer: "What is required for machine translation is not an accurate representation of 'meaning', whatever that may be, but an unambiguous representation of the entities involved in the discourse (objects, events, etc.) and the relations between them" (ibid.)

Phillips's focus on entities and relations already hints at *how* first-order formulae can conveniently encode propositional contents. A first-order SRL will have a *domain* containing precisely those "entities involved in the discourse". In formulae, the entities will thus be represented by *constants* and *variables*. Relations between them will be represented by *relation symbols* (cf. Ebbinghaus et al. (1994) for use of terminology).

Phillips's answer also raises a new question: What kinds of entities *are* involved in a discourse? In other words, what kinds of entities should the domain of a first-order SRL contain? And what kinds of relation symbols are needed? (2) shows two possible representations of the propositional content of the sentence (1). They differ considerably with respect to this question.

(1) A big boxer dates Mia.

(2) (a) $\exists b(\textsc{boxer}(b) \wedge \textsc{big}(b) \wedge \textsc{date}(b, \textsc{mia}))$

(b) $\exists e(\textsc{date}(e) \wedge \exists b(\textsc{boxer}(b) \wedge \textsc{big}(b) \wedge \textsc{subj}(b, e)) \wedge \textsc{obj}(\textsc{mia}, e))$

In (2a), all domain elements assumed are real-world individuals like people and objects. The contributions of common nouns, adjectives, and intransitive verbs to the propositional content are represented by one-place relation symbols, the contributions of transitive verbs by two-place relation symbols. The representation in (2b) assumes additional, more abstract domain elements: *events*. This allows for the use of *event variables* in a fashion known as "neo-Davidsonian" (Copestake et al., 1995, p. 19). One advantage of event variables is that they provide other subformulae with a means to refer to the event, offering an elegant way to flexibly add more information, as exemplified in (4).

(3) A big boxer dates Mia in the park on Sunday.

(4) $\exists e(\textsc{date}(e) \wedge \exists b(\textsc{boxer}(b) \wedge \textsc{big}(b) \wedge \textsc{subj}(b, e)) \wedge \textsc{obj}(\textsc{mia}, e) \wedge \textsc{place}(\textsc{park}, e) \wedge \textsc{time}(\textsc{sunday}, e))$

As Blackburn and Bos (2005, p. 47–50) argue, first-order logic is flexible enough to allow a great deal of freedom in choosing the domain elements and vocabulary for an SRL. The choice should be guided by the range of propositional contents one wishes to be able to express, and on the semantic theory one wishes to apply. Beyond events, it is perfectly possible to add situations or possible worlds to the domain, or points and spans in time, or to represent even properties, represented above as one-place relation symbols, as domain elements.

The freedom that Blackburn and Bose emphasize pertains particularly to first-order logic as a representational system, and to the use of inference tools such as theorem provers and model builders, which are generally applicable to formulae of all kinds of first-order languages. On the other hand, considerations of building semantic representations from natural language expressions, and of generating natural language expressions from semantic representations, depend on the choice of SRL a lot, as will be seen in the following sections.

```
s/love(X,Y)  --> np/X, vp/love(X,Y).
np/vincent   --> vincent.
np/mia       --> mia.
vp/love(X,Y) --> v/love(X,Y), np/Y.
v/love(_,_)  --> loves.
```

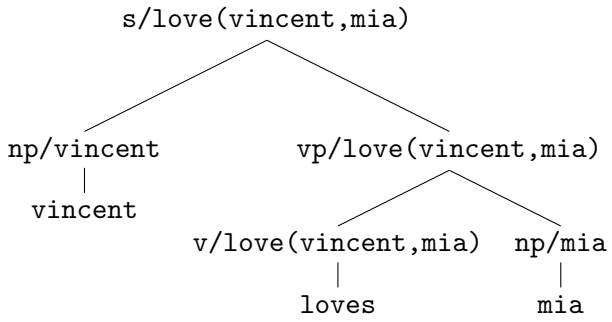

Figure 1: An example DCG grammar and an analysis tree for the sentence *Vincent loves Mia.*

2.2 Generation with Reversible Grammars

A distinction is commonly made between *strategic* NLG – deciding "what to say" – and *tactical* NLG – deciding "how to say it" (cf. van Noord, 1990, p. 141). With the construction of semantic representations (SRs), the strategic part can be expected to be covered, so I am concerned with tactical NLG only. Used as input for tactical NLG, first-order SRLs have played their most important role in a field one might call *Reversible Grammar Engineering* (RGE). This is a strand of research that views generation and parsing as two sides of the same problem: Constructing an analysis tree for a natural language expression. When the terminal nodes are given, it is called parsing – when the root node is given, it is called generation.

The set of admitted analysis trees is specified by a grammar, typically of a logic- or unification-based grammar formalism (cf. Shieber, 1988, p. 614). The non-terminal nodes of analysis trees are assumed to be complex structures and contain semantic information, typically in the form of expressions of a logical SRL. Figure 1 shows a miniature example grammar in the unification-based DCG formalism (cf. Covington, 1994, chapter 3) along with an analysis tree.

An important desideratum in RGE is that the same grammar be used for generation and parsing. The algorithms should be such that the process is reversible: Whenever parsing produces an analysis tree with a particular root node, generation from this root node should produce the original expression, and vice versa. However, the mapping is not required to be one-to-one. This would not be sensible to require, given that 1) many

natural language expressions are ambiguous, and 2) many propositional contents can be expressed in natural language in more than one way. Grammars for which a parser and a generator can be found such that the requirement of reversibility is fulfilled are called *reversible grammars*. For a motivation of RGE, see e.g. Kay (1975, p. 12) or Appelt (1989, p. 199 f.).

Two more characteristics of the grammar formalisms typically used in RGE deserve mention, as they influence the choice of SRL: The derivation of syntactic structure and of semantic representations are governed by the same set of rules, so they take place simultaneously. SRs are derived *compositionally*, i.e. the SR of any non-terminal node in the analysis tree (except the pre-terminals) is determined by the SRs of its child nodes. This implies that *every* node in an analysis tree (except the terminals) has an SRL. Thus the SRL must be able to represent the content not only of whole utterances to be generated, but also of smaller phrases.

2.2.1 Semantic Monotonicity

Further restrictions are often imposed on the grammar and on the SRL in order to make efficient generation possible. One such restriction is that of *semantic monotonicity*, introduced by Shieber (1988). It requires that in every analysis tree admitted by the grammar, the SR of each non-terminal node contains the SRs of all sub-nodes in a recognizable form (recognizable e.g. by string comparison or term unification). For illustration, consider the non-terminal rules of the semantically monotonic example grammar in figure 1: In each rule, each right-hand side SR unifies with a part of the left-hand side SR.

Shieber tackles the problem of parsing and generation as one of deductive proving: He regards a grammar as a set of axioms that can be used for constructively proving the grammaticality of a string (parsing) or the existence of a string matching some given SR (generation). He presents a common theorem-proving architecture that can be parameterized to be efficient for one process or the other. In either situation, the difficulty lies in making the constructive process *goal-directed* – it is not practical to construct every possible analysis tree and then check whether it conforms to the input requirements.

In parsing, goal-directedness is achieved e.g. by only considering lexical entries whose syntax matches the tokens in the input string. To make the generation process goal-directed, Shieber exploits semantic monotonicity. For example, faced with the task of constructing a tree with root `s/love(vincent,mia)`, it is possible to discard speedily any rule with a right-hand side SR that does not unify with any part of `love(vincent,mia)`, such as `s/snort(X) --> np/X, vp/snort`, `snort` being the offending SR.

2.2.2 Semantic Heads

The requirement of semantic monotonicity proved to be very restrictive with respect to certain linguistically plausible analyses. For example, the grammar in figure 2 is semantically nonmonotonic because one rule “swallows” the SR `up`. The “semantic-

```
s/pick_up(X,Y)       --> np/X, vp/pick_up(X,Y).
np/vincent           --> vincent.
np/mia               --> mia.
vp/pick_up(X,Y)      --> vp(up)/pick_up(X,Y), p/up.
vp(up)/pick_up(X,Y) --> v(up)/pick_up(X,Y), np/Y.
v(up)/pick_up(_,_)   --> picks.
p/up                 --> up.
```

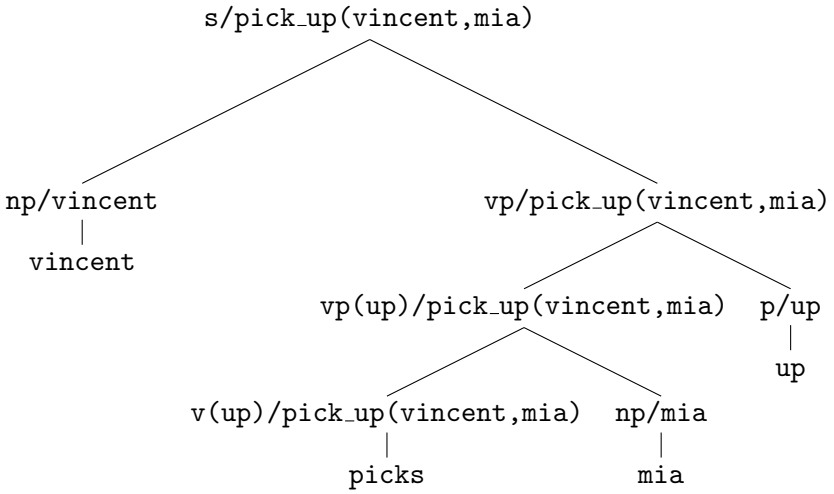

Figure 2: An example DCG grammar and an analysis tree for the sentence *Vincent picks Mia up*

head-driven generation" algorithm of Shieber et al. (1990) is both less restrictive and more restrictive than the algorithm of Shieber (1988): Not every right-hand side SR has to occur in the left-hand side SR of the respective rule, but in order to avoid highly non-deterministic top-down generation, there must be a *pivot* node as deep as possible in the analysis tree, ideally a leaf. A pivot node is defined as a node such that there exists a path from the root to it such that the SRs of all nodes on this path unify. The trick of the algorithm is to construct the pivot and its descendants as early as possible in order to have a great deal of structure already instantiated in subsequent top-down generation of the rest of the analysis tree.

Various extensions and improvements to semantic-head-driven generation are discussed in van Noord (1990). It quickly became "the single best-known algorithm for surface generation" (Reiter, 1994, p. 168), and apart from the original DCG implementation, it has been implemented at least for HPSG (Wilcock and Matsumoto, 1998). That it did not catch on in application-oriented systems (cf. Reiter, 1994, p. 168) may partly be due to a fundamental problem that is not limited to principled approaches like RGE, but still probably easier to avoid in less principled approaches: The problem of SR equivalence.

2.3 The Problem of SR Equivalence

In RGE, generators usually require that the input SR *could have been* derived from some natural language expression by the corresponding parser, working with the same grammar (cf. Phillips, 1993, p. 214). This natural language expression is then in the output of the generator. To say it pointedly, generation algorithms for reversible grammars seem to be specialized in an exercise with no practical value. In practical applications, the input to a tactical generator does not come from a corresponding parser. More likely, it is produced

- by a strategic generator or "reasoner" of some sort (cf. Shieber, 1993, p. 180), or
- in interlingua MT, by a parser that necessarily works with a different grammar (that of the source language), or
- in transfer MT, by a transfer component from the output of the source language parser (cf. Phillips, 1993, p. 214).

For such components, it is difficult to ensure that their output is, in its exact form, admitted by the grammar that the tactical generator works with. Even seemingly superficial and insignificant differences can thwart generation. In first-order logic, such "insignificant" differences are exemplified in (6) – both formulae are logically equivalent, but differ in form due to the nesting of quantifier scopes and the order of the arguments to conjunction. If a reversible grammar associates the sentence (5) with the SR (6a) but not with (6b), feeding (6b) to the generator will not produce the desired output.

(5) A boxer loves a woman.

(6) (a) $\exists b(\exists w(boxer(b) \wedge woman(w) \wedge love(b, w)))$
(a) $\exists w(\exists b(woman(w) \wedge boxer(b) \wedge love(b, w)))$

This problem is in fact a very fundamental one and goes by many names, like the "subset problem" (Landsbergen, 1987, p. 129), "the completeness of the generator with respect to the logic" (Phillips, 1993, p. 214), or "the problem of logical-form equivalence" (Shieber, 1993). Note that the problem is not specific to the reversible grammar scenario – it occurs whenever the SRL contains equivalent expressions that should be treated alike by the generator. An apparent solution is to introduce a new step into the pipeline, before tactical generation, to resolve SR equivalence, e.g. by converting the SR to a "canonical form" that the generator, or its grammar, is attuned to and guaranteed to cope with. For first-order logic, this is not feasible because equivalence of first-order formulae is an undecidable problem.

Note, however, that logical equivalence need not be a good approximation to meaning equivalence anyway. For example, the two formulae in (8) are equivalent, but arguably, only the former captures the content of sentence (7) adequately.

(7) Butch snorts and either Mia is a boxer or Mia is not a boxer.

(8) (a) $\textsc{snort}(\textsc{butch}) \wedge (\textsc{boxer}(\textsc{mia}) \vee \neg\textsc{boxer}(\textsc{mia}))$
(b) $\textsc{snort}(\textsc{butch})$

For a brief discussion of such philosophical problems in the context of SR equivalence, see Shieber (1993, p. 187 f.).

2.3.1 Solution 1: Minimal Structure

Due to the undecidability of equivalence in first-order logic, many efforts have been directed towards generation algorithms for SRLs based on weaker variants of first-order logic, SRLs for which a useful and efficiently computable notion of equivalence can be found. Phillips (1993) and Kay (1996) are representatives of this approach. The SRLs their generation algorithms work with are characterized by minimal structure, such as to avoid spurious ambiguity due to nested scopes, commutativity and asscociativity. Minimal structure is achieved by opting for a "neo-Davidsonian" representation as in (2b) and essentially restricting formulae to being conjunctions of simple formulae built exclusively from relation symbols, constants, and variables. (Phillips and Kay call variables "indices".) All occurring variables can be regarded as implicitly existentially quantified over. The order of the conjuncts is regarded as immaterial. The following example SR for the sentence *John wrote a letter* is from Phillips (1993, p. 223):

(9) $\textsc{john}(j)\&\textsc{write}(e)\&\textsc{letter}(l)\&\textsc{past}(e)\&\textsc{actor}(e, j)\&\textsc{patient}(e, l)$

The SRLs Phillips and Kay use remain purely exemplary and expositional, as the two papers are primarily concerned with their respective generation algorithms, both chart-based and optimized using lexical information from the SR. The idea of an SRL with

minimal structure is made more explicit by Copestake et al. (1995), who propose a "meta-level language" for SRLs called Minimal Recursion Semantics (MRS). They also note the need for explicitly representing the scopes of universal quantification, negation, disjunction, etc. They do this while retaining minimal structure by representing scope numerically, which at the same time offers an elegant way to represent underspecification, as in the following representation for two readings of the sentence *Every dog chased some cat*, taken from Copestake et al. (1995, p. 20):

(10) $\text{EVERY}_1(x, 3, n), \text{DOG}_3(x), \text{CAT}_7(y), \text{SOME}_5(y, 7, m), \text{CHASE}_4(e, x, y)$

After an introduction to their framework, Copestake et al. withdraw from the field of first-order-like representations and turn to feature structures. But even their expository approach, thanks to its sophistication and conciseness, bodes well for the possibility of principled and efficient generation from first-order-like representations.

2.3.2 Solution 2: Isomorphic Grammars

A completely different solution to the problem of SR equivalence was chosen in the Rosetta project, a long-term effort to build an MT system based on Montague Grammar (Landsbergen, 1987, p. 113 f.). The original approach was to use reversible grammars as outlined above for all source and target languages, in fashion strongly influenced by the work of Richard Montague. The SRL used as an interlingua was, in this case, a language of intensional logic. The grammar formalism used was a variant of that described in Montague (1973), modified to have better computational properties and called *M-grammars* (Landsbergen, 1987, p. 120–124). The problem of SR equivalence manifested itself in the fact that not all of the grammars could be guaranteed to map natural language expressions to the same subset of intensional logic, unless the grammars were developed in close conjunction. This measure would have relinquished the main advantage of interlingua MT: inter-independence of the modules for the individual source and target languages.

The approach eventually taken in the Rosetta project does require this sacrifice, but also offers a reward: It is not necessary to worry about adequate semantic representations at all. SRs were discarded, instead the grammars used were made *isomorphic*. In this quite unique approach, for each pair of grammars, each rule in one grammar corresponds to (at least) one rule in the other grammar (Landsbergen, 1987, p. 131–137). Parsing of the input expression then amounts to determining which rules of the source grammar could have been applied to derive this expression, and deriving the output expression amounts to applying the corresponding rules of the target grammar. The downside is that isomorphy is a very tough requirement, even if all grammars are developed together. According to Copestake et al. (1995, p. 16), it leads to unnatural analyses and is not really feasible for multiple languages.

2.4 Are Logical Formulae Adequate for Translation?

Landsbergen's (p. 128 f.) three reasons for discarding Montagovian intensional logic as an interlingua form a very concise account of some, if not all fundamental issues in MT with logical formulae. For the most part, they pertain to logical SRLs in general. What are their implications for the present thesis?

One of Landsbergen's reasons is the problem of SR equivalence, which he calls the "subset problem". Within the field of MT with a purely semantic interlingua, it appears that this problem can only be solved by relatively sophisticated techniques and, more importantly, restricting the SR to something less expressive than first-order logic. This has not been done for the SRL used by BB1, but for the endeavor at hand, there is hope that the fragment of English dealt with and the set of logical formulae produced is so small that the problem will come up only in a restricted form that can be dealt with rather ad-hoc.

Another of Landsbergen's reasons for discarding Montagovian intensional logic is that it represents only "meaning in the model-theoretic sense", i.e. propositional content, and leaves out "information on pragmatic and stylistic aspects" which may be relevant for translation. While this pertains to Montagovian intensional logic and most logical SRLs used in expository material, the representational flexibility of first-order logic mentioned in section 2.1 suggests that, once the pragmatic and stylistic information relevant to translation is identified, it could also be encoded in the SRL. But how this should be done in first-order or even in more restricted logics is beyond the scope of this thesis, which does restrict itself to propositional content.

Finally, there is an argument against semantic representations in MT in general: To state and exploit translational equivalences between words and constructions of two languages, it is not always necessary to give an adequate account of their semantics. For example, the difficulties of describing the semantics of an intensional verb like *believe* should not impede the process of translating it to Dutch, where the verb *geloven* serves the same purpose. This is a very valid point but of course does not pertain to the present effort, where it is an explicit aim to explore possibilities of MT in the same "medium" as CS, with semantic representations.

2.5 The Semantic Representation Language of BB1

Copestake et al. (1995, p. 18) name three requirements that an SRL suitable for Machine Translation should fulfill: It should not have much structural complexity, it should allow for underspecified representations, and it should support inference. Only this last requirement is fulfilled by the first-order SRL of BB1. Underspecified representations are used internally in derivation but not in the output formulae. Finally, the structure is not "flat" but rather inclined towards nesting. Consider the logical formula (11) and its Prolog representation in (12), representing one reading of the sentence *Every big boxer that does not kill a robber loves a woman.* Every quantifier and boolean connective introduces a level of nesting. In particular, conjunctions are far from being flat unordered

sets as in (10) – the boolean connective `and` is binary, thus conjunctions of multiple formulae are multiply nested.

(11) $\forall x(((\text{BOXER}(x) \wedge \neg\exists y(\text{ROBBER}(y) \wedge \text{KILL}(x, y))) \wedge \text{BIG}(x)) \rightarrow \exists z(\text{WOMAN}(z) \wedge \text{LOVE}(x, z)))$

(12)
```
all(G, imp(and(and(boxer(G), not(some(S, and(robber(S), kill(G,
S))))), big(G)), some(X, and(woman(X), love(G, X)))))
```

Despite the structural complexity, it is still a simple language, easy to overlook intuitively. For the exercise at hand, to generate German from the SRL, it is presumably more straightforward to hand-craft rules that turn formulae into German step by step than trying to write a reversible grammar for German that maps to the same SRL. An implementation of the former approach will be described in the next section.

For an account of first-order logic, the way Blackburn and Bos use it to represent meaning, and how it is represented in Prolog, see Blackburn and Bos (2005, p. 1–18, 32 f.). Due to the application-oriented nature of the remainder of this thesis, preference will be given to the Prolog notation for formulae as in (12).

3 GG1: Architecture and Implementation

I have implemented an NLG component for generating German sentences from the kind of logical formulae produced by BB1, the software of Blackburn and Bos (2005). In honor of their work, I called the generation component GG1 (*GG* for *generating German*). It provides a user interface for translating sentences from a certain fragment of English to a corresponding fragment of German. BB1 is used to derive logical formulae from the English sentences; subsequently, German sentences are generated from the logical formulae. In this section, I describe the architecture and implementation of GG1, focusing on linguistically relevant details. Analyses of and judgements about German sentences are based on my own background knowledge and intuition, unless otherwise stated.

GG1 is organized into five Prolog modules. At the top level there is translate, which provides the glue between BB1 and the generation code. The predicate translate/2 delegates the construction of semantic representations to the holeSemantics module of BB1[1] and collects the results, which may be multiple formulae due to semantic ambiguity. For each formula, it tries to generate a German sentence. The predicate translate/0 provides the user interface, allowing users to type in an English sentence and showing them the German translation(s), numbered, punctuated, capitalized, and filtered such as not to display duplicates. (Duplicates may arise when German translations show the same ambiguity as the original English sentence.)

[1] holeSemantics is the last in a series of steps in which Blackburn and Bos develop and explain key ideas of building semantic representations, becoming gradually more complex. holeSemantics was chosen over the other, less complex modules, because it accounts for scope ambiguities in English most exhaustively. The proper treatment of scope was of special interest to generating German (see section 3.5).

$$
\begin{aligned}
S_{SVO} &\rightarrow DP \quad VP_{SVO} \\
S_{SOV} &\rightarrow DP \quad VP_{SOV} \\
DP &\rightarrow D \quad NP \\
NP &\rightarrow N \\
NP &\rightarrow NP \quad CP \\
NP &\rightarrow AP \quad NP \\
CP &\rightarrow D_{rel} \quad S_{SOV} \\
AP &\rightarrow A \\
VP_{SVO} &\rightarrow V \quad DP \quad Neg \quad Part \\
VP_{SOV} &\rightarrow DP \quad Neg \quad PartV \\
Neg &\rightarrow \varnothing \\
Neg &\rightarrow nicht
\end{aligned}
$$

Figure 3: Sketch of the grammar assumed for German

The rules module is where generation from first-order formulae takes place and where knowledge about German syntax and its relation to first-order SRs is encoded. A few helper predicates that it uses to manipulate data structures are collected in the util module, others are imported from the comsemPredicates module of BB1. Lexical information is provided by the lex module which maps atomic symbols from the SRL and sets of morphosyntactic features to forms of German words. It does so partly in a hard-coded fashion, partly via lexical rules describing inflection and taking information about word stems and their semantics from the stems module.

3.1 From Semantics to Syntax

GG1 produces German sentences with an active voice, present tense, third person singular verb and a subject. If the verb is *ist* ("is"), there is also a predicate noun. If the verb is transitive, there is also an accusative object. Subjects, objects, and predicate nouns are always either a proper name, an existentially quantified determiner phrase, or a universally quantified determiner phrase. Determiner phrases consist of a determiner and a noun phrase. A noun phrase contains one noun and may contain any number of adjectives and relative clauses. Finally, the negation marker *nicht* may occur.

A simple model of German syntax underlies the generation methodology of GG1. This model mainly serves as a tool for enforcing linguistic constraints in a reasonably systematic way. No special consideration is therefore given to the adequacy of the model with respect to any theory of German syntax. Figure 3 shows a sketch of the assumed

underlying phrase structure grammar, using widely known atomic labels for non-terminal nodes (such as S, DP, VP). The grammar is binary-branching for the most part, except for verb phrases which have a flat structure.

At no point in the code is the assumed phrase structure grammar declared explicitly, but it is manifest in that for each phrasal category, there is (at least) one Prolog predicate in the rules module that corresponds to it. For example, the s/5 predicate generates sentences; vp_intrans/6 generates intransitive verb phrases, vp_trans/7 generates transitive verb phrases, and so on. Generation proceeds by **recursive descent**: Predicates corresponding to one category call predicates corresponding to its subcategories, passing down information about both semantic and morphosyntactic features of the phrases to generate. This is depicted in figure 4 by a partial simplified proof tree. The nodes of the tree represent the respective predicates, with arguments instantiated as at the point of calling. By looking at the underlined parts, you can follow the semantic information as it disperses across the different phrases.

The semantic information initially passed to s/5 is a whole logical formula as produced by BB1. The semantic information passed to predicates generating smaller phrases is created by progressively disassembling the formula into its parts. For example, vp_trans/5 has three semantic arguments: One for the *symbol* of the verb, one for a constant or a quantified term (see section 3.2) corresponding to the accusative object, and one indicating whether the verb phrase to be generated should be negated. *Symbols*, like boxer, collapse, or blue, are the smallest meaningful parts of the formulae. They can be looked up in the lexicon to obtain the corresponding content words.

In general, the phrase-generating predicates have between one (s/5, dp/5) and five (np/10) arguments for semantic information, followed by a number of arguments for morphosyntactic information, followed by the final argument that contains the generated phrase after the predicate succeeds. The semantic and morphosyntactic features are usually fully instantiated when a predicate is called: The semantics of a phrase to generate is pre-determined by the formula; its morphosyntactic features must match the syntactic context determined by its ancestor phrases. The most important exceptions to this rule are Prolog variables representing logical variables – these never get instantiated – and the arguments for sex and gender (see section 3.4) – these mostly do not get instantiated before lexical lookup.

The generating predicates do not generate strings (or flat lists of words) directly. Rather, sentences are constructed as syntactic trees corresponding to the assumed grammar, represented as Prolog terms. The main reason for this design decision is to preserve syntactic information for use in post-processing. For example, punctuation is performed based on where phrases with the label CP begin and end.

3.2 Unscoping Quantifiers

The most significant structural difference between the English sentences that BB1 can process and the corresponding logical formulae is how quantification is represented: In the fragment of English, quantification is indicated by determiners like *a* (for existential quantification) or *every* (for universal quantification) in noun phrases. A noun phrase

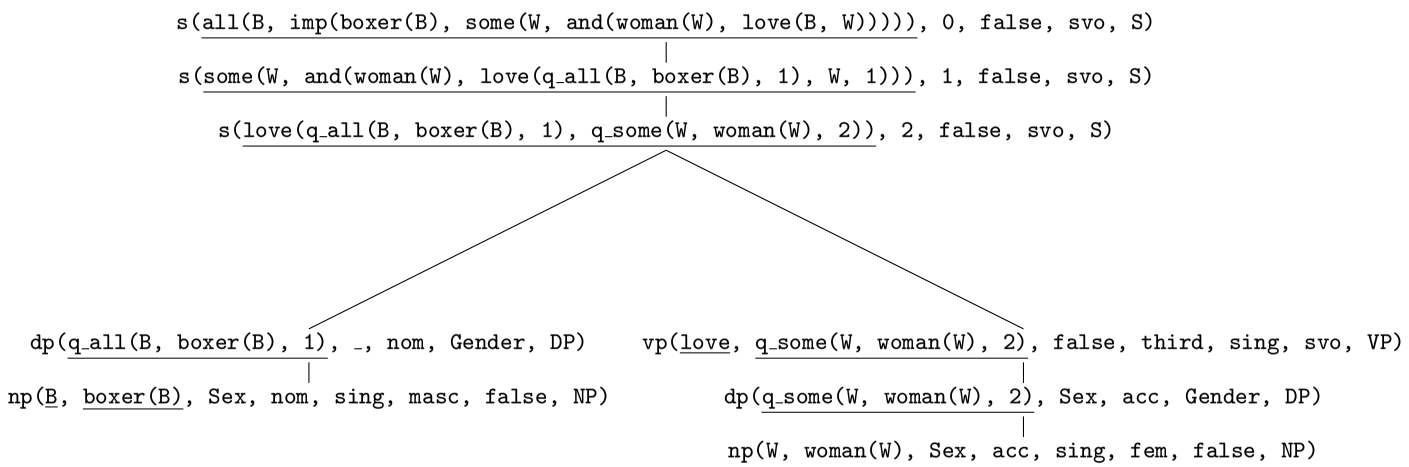

Figure 4: Partial simplified proof tree for generating the sentence *Jeder Boxer liebt eine Frau.* Only calls to phrase-generating predicates are shown. The semantic arguments are underlined

with such a determiner is called a *quantified noun phrase* and realized, like other noun phrases, at a position determined by the syntax of the verb whose argument it is. By contrast, in logical formulae, quantifiers are realized outside of the argument slots of verbs. The argument slots are filled by placeholder variables bound by quantifiers. This allows to express quantifier scope unambiguously. For example, two readings of the sentence (13) are shown in (14). Possibilities to express each reading unambiguously in English are shown in (15). Such possibilities exist also in German, but the question of when they are appropriate in everyday language and when they sound markedly formal is too complex to consider in this thesis. The scope of quantifiers is therefore always reduced again to "in-place" quantified noun phrases as in (13), resulting in translations which are again ambiguous, like (17).

(13) Every boxer loves a woman.

(14) (a) `all(X, imp(boxer(X), some(Y, and(woman(Y), love(X, Y)))))`
(b) `some(X, and(woman(X), all(Y, imp(boxer(Y), love(Y, X)))))`

(15) (a) For every boxer there is a woman that he loves.
(b) There is a woman that every boxer loves.

(16) (a) `love(q_all(B, boxer(B), 1), q_some(W, woman(W), 2))`
(b) `love(q_all(B, boxer(B), 2), q_some(W, woman(W), 1))`

(17) Jeder Boxer liebt eine Frau.

To bridge the gap between logical formulae and syntax with respect to quantifiers, a first step is to change the formulae to hold *unscoped* representations of quantification, as in (16). These are produced as follows: When given a quantified formula, the sentence-generating predicate `s/5` extracts the restriction and the nuclear scope from the scope of the quantifier and generates a *quantified term* (QT) from the restriction. QTs were used, in a different notation, by Alshawi et al. (1991, p. 162) in their SRL called *Quasi Logical Form*. In GG1, a quantified term is a term whose functor is one of `q_some`, `q_all`, `q_no`, and `q_notall`, chosen depending on the type of quantification and on whether the formula was negated. Every quantified term has three arguments: One for the variable quantified over, one numerical argument indicating the order in which this quantifier was unscoped (thus retaining scope information), and one for the restriction. The QT is then substituted for the variable quantified over in the nuclear scope, and the result is the new formula, which is recursively passed to `s/5` again.

The upper part of the proof tree in figure 4 shows how unscoping gets applied to (14a) twice to yield the representation in (16a). This is closer to the syntactic arrangement to be generated and can be processed by those rules of `s/5` that proceed with the actual generation of the syntax tree. Note that the unscoped form of (14b), shown in (16b), is almost the same, only the numbers indicating the order in which quantifiers take scope are swapped. From this point on, the generation process is exactly the same for both

formulae, as the numbers are consulted only in certain cases involving negation (see section 3.5).

One technique deserves explanation, namely how the scope of a quantifier is dissected into restriction and nuclear scope. In the case of universal quantification, this is trivial because the scope of a universal quantifier is an implication where the antecedent is the restriction and the consequent is the nuclear scope. (This does not necessarily hold for formulae denoting propositions about all domain elements, like `all(X, boxer(X))`, but these do not seem very relevant for natural language and are in any case not produced by BB1.) In the case of existential quantification, this is not so simple. Admittedly, in formulae produced by BB1, the scope is always a binary conjunction where the restriction is the first and the nuclear scope is the second argument. But relying on this superficial property is not desirable if at least some generality with respect to first-order formulae is aimed for. The nuclear scope is therefore found by a check implemented in the dissect/3 predicate that accepts as a nuclear scope any term whose functor is one of all, some, not, eq, or the symbol of a verb entry in the lexicon. These are the terms from which, after substitution of free variables by quantified terms, sentences can be generated.

3.3 The Lexicon

Word forms are an important resource to the generation process: They are the building blocks of the string that will eventually be presented to the user. The generation process must choose word forms according to semantic and morphosyntactic features. It is the purpose of the lex module to map combinations of semantic and morphosyntactic features to word forms.

For the most part, the lexicon consists of lexical entries in the form of Prolog facts. The predicates that these facts belong to are organized like the phrase-generating predicates in the rules module. For every part of speech category (and for certain subcategories) there is a predicate: n_common/6 for common nouns, n_proper/4 for proper names, v_be/4 for the verb *sein* ("to be"), v/5 for other verbs, d_indef/5 for indefinite articles, d_indef_neg/5 for negative indefinites, d_univ/5 for the determiner *jede* ("every"), d_univ_neg/5 for its negated version, *nicht jede* ("not every"), d_rel/6 for relative pronouns, and a/6 for adjectives.

The last argument of a lexical entry contains a tree fragment with a preterminal (the part-of-speech symbol) and a terminal (the word form). Common nouns, proper names, verbs other than *sein*, and adjectives are *content words*. Every content word corresponds to an atomic *symbol* in the SRL of BB1. This symbol is the first argument of a lexical entry. Nouns also have a semantic argument for sex (see section 3.4). The rest of the arguments contain morphosyntactic features by which words are selected according to morphosyntactic context.

The morphosyntactic features of nouns distinguished here are case (nom or acc), number (currently always sing), and gender (masc, fem, or neut). Determiners and adjectives additionally have a feature indicating whether a particular form is inflected according to the "strong" declension (true or false). Verbs have person and number (currently always third and sing). The word forms of verbs are represented a bit differently in order

to accomodate *particle verbs*, which can split into two parts depending on the syntactic context. The second-to-last argument of a verbal lexical entry contains the simplex form, the last one may contain a particle.

Since there is a separate lexical entry for every possible combination of semantic and morphosyntactic features, the lexicon is represented in a very redundant way. For example, all words except for verbs have at least two entries that differ only in the case feature (nom vs. acc). It would be relatively easy to capture systematic occurrences like these in *lexical rules*, thus representing the lexicon in a more compact and easily extensible way. For adjectives, where the number of morphosyntactic feature combinations relevant to GG1 is no less than twelve, this measure has in fact already been taken. A separate module called stems maps the semantic symbols to base forms of adjectives, and twelve lexical rules in the lex module append the right inflectional ending depending on morphosyntactic features.

3.4 Sex, Gender, and the Verb "ist"

All German nouns have grammatical gender – they are masculine, feminine, or neuter. Adjectives and determiners agree with the gender of their noun. Thus, the lexicon predicates for nouns, adjectives, and determiners have a morphosyntactic argument for gender whose value is one of fem, masc, and neut. In addition, most nouns that refer to persons have a "sex feature", i.e. they imply that the persons they refer to are of a particular sex. This is obvious in cases like "Frau" (*woman*), which only refers to female persons, or "Mann" (*man*), which only refers to male persons. Apart from that, there are many pairs of words like *Boxerin*/*Boxer* ("boxer"), where *Boxerin* only refers to female boxers and *Boxer* (usually) only refers to male boxers. To reflect this, the lexicon predicates for nouns have a semantic argument for sex which is either masc or fem – or none, if a noun has no preference concerning the sex of the referent. The latter is, of course, the case for all nouns denoting inanimate entities, and also for some denoting persons, like *Krüppel* ("gimp").

Unlike case or number, gender and sex are features that nouns have independently of their inflectional forms. Because of this, the corresponding arguments of phrase-generating predicates (cf. section 3.1) are not instantiated before the word form is looked up in the lexicon. After the lookup, the predicates np/10 and dp/5 enforce the constraint that adjectives and determiners match the gender of the corresponding noun.

In certain cases, the sex feature needs to be matched as well. The SRL of BB1 contains the two-place relation symbol eq/2 that is special with respect to the logic in that it denotes *equality* between its arguments (cf. Blackburn and Bos, 2005, p. 17, 32). With respect to generation, it is not much different from relation symbols representing transitive verbs, except that the verb *ist* ("is") has a *predicate noun* in nominative case instead of an object in accusative case.

Under certain conditions, it is desirable for the sex of the subject to match the sex of the predicate noun. For example, the formula (18), stating that Mia is a boxer, should be rendered as (19a) but not as (19b) because the lexicon specifies Mia's sex as female, thus the female translation of *boxer* is appropriate. On the other hand, for seemingly

absurd formulae like (20) and (22) it is okay to receive absurd translations where the sexes do not match – failing to generate (21) or (23) is not desirable.

(18) `some(X, and(boxer(X), eq(mia, X)))`

(19) (a) Mia ist eine Boxerin.
(b) * Mia ist ein Boxer.

(20) `some(X, and(woman(X), eq(vincent, X)))`

(21) Vincent ist eine Frau.

(22) `some(X, and(qpwc(X), eq(vincent, X)))`

(23) Vincent ist ein Hamburger Royal.

The desired behavior is achieved with a relatively simple technique: When generating a sentence with *ist*, s/5 requires that the Sex arguments of the generated subject DP and of the generated VP can be unified – but enforces this requirement only if it does not rule out all possible solutions. This way, *Boxerin* wins against *Boxer* as a noun to apply to Mia, but *Frau* ("woman") can be applied to Vincent since there is no entry with male sex for the symbol woman.

The current setup, where pairs of words with a sex preference like *Boxer/Boxerin* are represented as separate lexical entries with the same semantic symbol (boxer) leaves some room for improvement. Consider the English sentence *Every boxer snorts.* It asserts that every boxer, regardless if male or female, snorts. Its semantic representation `all(X, imp(boxer(X), snort(X)))` is just as neutral. Unfortunately, GG1 currently treats this formula as if it were ambiguous between "Every female boxer snorts" and "Every male boxer snorts", offering the two translations in (24), but not a single one that captures the meaning correctly. Arguably, (24b) can be interpreted as a "generic male", but nowadays usage tends to avoid this for reasons of gender neutrality. A more politically correct and unambiguous rendering would be (25). An improvement to be envisioned is thus to generate such "slashed" forms automatically, preferrably in generation rather than in post-processing for efficiency. This would require some way of percolating among generating predicates feature sets that are underspecified with respect to sex, gender, and strong vs. weak inflection.

(24) (a) Jede Boxerin prustet.
(b) Jeder Boxer prustet.

(25) Jede/r Boxer/in prustet.

3.5 Negation

Besides quantifiers, negation too gives rise to scope ambiguities in English. The holeSemantics module of BB1 is very thorough in accounting for all possible readings. For example, the sentence (26) receives no less than six readings, shown in (27), of which however some are logically equivalent. In (a) and (b), there is no boxer whatsoever that loves any woman whatsoever. This is the logical inverse of the sentence *A boxer loves a woman.* (c) asserts that there is some boxer who does not love any woman. (d) and (f) assert that there is some woman that some boxer does not love. (e) asserts that there is some woman that no boxer loves.

(26) A boxer does not love a woman.

(27) (a) `not(some(G, and(boxer(G), some(N, and(woman(N), love(G, N))))))`
(b) `not(some(N, and(woman(N), some(G, and(boxer(G), love(G, N))))))`
(c) `some(G, and(boxer(G), not(some(N, and(woman(N), love(G, N))))))`
(d) `some(G, and(boxer(G), some(N, and(woman(N), not(love(G, N))))))`
(e) `some(N, and(woman(N), not(some(G, and(boxer(G), love(G, N))))))`
(f) `some(N, and(woman(N), some(G, and(boxer(G), not(love(G, N))))))`

The only form of negation in English that BB1 deals with is *sentential negation* with one of *is not* and *does not*. The syntactically corresponding type of negation in German, with the marker *nicht*, does not convey as many readings. In the cases where negation applies to a quantified formula, such as in (27a,b,c,e), it must be realized in the corresponding quantified DP. In the case of existential quantification, this can be done by replacing the indefinite article *ein* by the negative indefinite *kein*. Thus, the German renderings for (27) are as follows:

(28) (a) Kein Boxer liebt eine Frau.
(b) Kein Boxer liebt eine Frau.
(c) Ein Boxer liebt keine Frau.
(d) Ein Boxer liebt eine Frau nicht.
(e) Kein Boxer liebt eine Frau.
(f) Ein Boxer liebt eine Frau nicht.

Note that in (28b) negation is realized in the *boxer* DP instead of the *Frau* DP although in (27b) it is the quantified formula corresponding to *woman* that is negated. To obtain (28b), which is far more easily understood as meaning (27b) than *Ein Boxer liebt keine Frau*, one additional trick was required: If a "positive" QT (with functor q_some or q_all) ends up in the subject position of a transitive verb and a "negative" QT (with functor q_no or q_notall) ends up in the object position and the negative object QT has wider scope than the positive subject QT, then negation is "moved" to the subject by

replacing the functors of the QTs by their "inverse". This behavior is implemented by the adjustTransNeg/4 predicate in the rules module.

The following example derivation shows the relevant steps for (27b):

```
not(some(N, and(woman(N), some(G, and(boxer(G), love(G, N))))))
Unscoping yields:        some(G, and(boxer(G), love(G, q_no(N, woman(N), 1))))
Unscoping yields:        love(q_some(G, boxer(G), 2), q_no(N, woman(N), 1))
Moving negation yields:  love(q_no(G, boxer(G), 2), q_some(N, woman(N), 1))
```

Just like in the English original, some of the readings in such constructions depend heavily on stress and context in order to be understood. This is particularly true when universal quantification is involved. In a sentence-based translation system like this (i.e. there is no context) that focuses on propositional content (i.e. there are no style issues to consider), it would be best to generate sentences that convey the respective reading as overtly as possible. (29) and (30) show examples of formulae, the way they are currently rendered by GG1 and how the content could be expressed more clearly in German. Defining such cases clearly and implementing the additional syntactic operations (e.g. fronting) has to be left for future work.

(29) (a) **Formula:** `not(all(N, imp(woman(N), some(G, and(boxer(G), love(G, N))))))`

(b) **Current rendering:** Ein Boxer liebt nicht jede Frau.

(c) **Suggestion:** Nicht jede Frau liebt ein Boxer.

(30) (a) **Formula:** `all(N, imp(woman(N), not(some(G, and(boxer(G), love(G, N))))))`

(b) **Current rendering:** Kein Boxer liebt jede Frau.

(c) **Suggestion:** Kein Boxer liebt eine Frau.

(31) (a) **Formula:** `all(I, imp(restaurant(I), not(know(mia, I))))`

(b) **Current rendering:** Mia kennt jedes Restaurant nicht.

(c) **Suggestion:** Mia kennt kein Restaurant.

4 Evaluation

4.1 Fundamental Problems

Machine Translation systems are commonly classified along various dimensions – see for example Hutchins and Somers (1992, chapter 4) or Carstensen (2008, chapter 10) for an overview. Among other dimensions, systems can be classified by the workflow between human and machine, the number of supported language pairs, and the depth of processing.

The sytem presented in this thesis is a representative of **Fully Automatic Translation (FAT)**, as opposed to Machine-Aided Human Translation (MAHT) or Human-Aided Machine Translation (HAMT), since there is no interaction between user and

system beyond the input of a sentence and the output of possible translations. At present, it is **bilingual** rather than multilingual and **monodirectional** rather than **bidirectional** because only translation from English to German is supported. The fact that there is only one language pair and one direction of translation could, in theory, make it hard to tell whether it is an interlingual or a transfer-based system because processing is a single pipeline from one language to another, with no points that are shared between multiple language pairs and could be identified as the points where analysis ends and synthesis starts, or where transfer starts or ends. In practice, however, it is clear that the interface between BB1 and GG1 is meant as the interlingual point in the processing architecture. For translating to a different target language, one would have to replace GG1 by a corresponding generation component. So the answer is: The system is **interlingual**.

Whether the interlingua used is good is another question. As noted in section 2.5, the SRL used as an interlingua has a relatively rich structure which makes it difficult to solve the problem of SR equivalence. The pragmatic approach taken in GG1 takes various measures to tackle it, but a complete solution is not in sight. The prime example is the structure of conjunction: In the SRL conjunction is always represented in a binary-branching fashion, which happens to mirror the syntactic analysis of the original English sentence closely. Consider the sentence in (32) and its SR in (33). The structure of the SR betrays the order of the two adjectives in the sentence, although this is immaterial to the propositional content. Since, as Landsbergen (1987, p. 128) argues, "making use of the form of logical expressions is (...) in conflict with the spirit of Montague Grammar" or of any framework where content should be conveyed purely semantically, GG1 repudiates the structural information and puts the formulae representing the meaning of the adjectives into a list. (The same happens, in `np/7`, for formulae representing relative clauses, and for formulae representing nouns.) The order of the adjectives is then chosen nondeterministically, leading to the two translations in (34). Thus, the strategy for dealing with SR equivalence in nested conjunctions is to flatten them, explicitly discardig information that does not contribute to the propositional content.

(On a side note, (34) illustrates the third of Jan Landsbergen's reasons against using logical SRLs rendered in section 2.4: (34b) is a completely adequate translation of (32), and it is not necessary to have an alternative translation with a different ordering of adjectives, so it would be wise to stay close to the source syntax in translating. A counter-argument could be that if the order of adjectives has any significance in English, then in a more sophisticated semantic system the interlingua should be capable of expressing this significance, and the generator would be concerned with expressing this significance in a way appropriate to the target language. This way could *happen* to be the same order.)

(32) Butch is a big blue boxer.

(33) `some(G, and(and(and(boxer(G), blue(G)), big(G)), eq(G, butch)))`

(34) (a) Butch ist ein blauer großer Boxer.

(b) Butch ist ein großer blauer Boxer.

Nested conjunctions are by far not the only way in which first-order formulae can be equivalent. Another example that emerges as a practical problem in GG1 is the interplay between negation and quantification, discussed before in section 3.5. Recall that the formulae in (35) are logically equivalent – meaning "There is no boxer that loves any woman" – and best translated to German as (36a). Unfortunately, GG1 renders (35b) as (36b).

Similarly to flattening nested conjunctions, this particular problem could probably be fixed without too much effort by some clever ad-hoc measure. But in general, the problem remains that the semantics of formulae is hidden from the system, and for this reason, it is constantly in danger of missing the optimal target language rendering. This is because GG1 analyzes logical formulae in a purely syntactical fashion and does not perform any inference. As mentioned before, for first-order logic, the equivalence problem is undecidable. Also, inferencing is arguably not the job of a tactical generator. The call for an SRL where inferring equivalence is tractable (cf. section 2.3.1) is becoming more and more understandable.

(35) (a) `not(some(G, and(boxer(G), some(N, and(woman(N), love(G, N))))))`

(b) `all(N, imp(woman(N), not(some(G, and(boxer(G), love(G, N))))))`

(36) (a) Kein Boxer liebt eine Frau.

(b) Kein Boxer liebt jede Frau.

Another problem of the interlingua is that its lexicon mirrors the fragment of English BB1 deals with one-to-one, carrying over all ambiguities and not making certain distinctions that would be needed for translation to German. The word *plant* ("Pflanze" vs. "Fabrik") is such an example, and so is the transitive verb *shoot* which denotes different relations between individuals depending on what its object is – shooting a gun is not doing the same thing to a gun that you do to a robber when shooting a robber. In an MT system with a truly unambiguous interlingua, such disambiguations would have to be made in analysis, but generation faces complex challenges as well, for example structural idiosyncrasies of the target language. As a simple example, the two-place relation date in the SRL has been translated somewhat insufficiently with the transitive verb *(jemanden) treffen* instead of e.g. the syntactically more complex *mit (jemandem) ausgehen*. These problems show how quickly one runs into the "big issues" of MT, the problems of dealing with ambiguities and differences between languages (Hutchins and Somers, 1992, p. 99–106) even with a system that translates just between toy fragments of two relatively similar languages.

4.2 Possible Improvements of Details

This said, certain loose ends will certainly stay loose, but other aspects may be considerably improved in future work. As already envisioned in section 3.4, the capability to

generate "artifical" forms like *jede/r große Boxer/in* would help to give more accurate translations for gender-neutral English forms like *every big boxer*. Also, the adjectives *male* and *female* could be added to the lexicon and treated specially in generation – for example, *a female boxer* would then simply be translated as *eine Boxerin*.

An improvement envisioned in section 3.5 concerns the interplay of quantification and negation, where there is potential to give more natural and less ambiguous German translations than currently produced.

A handful of syntactic phenomena from BB1's fragment of English, namely conditional sentences, disjunction, and prepositions, cannot yet be translated because they lack counterparts in the generator. Support for more complex lexical entries, such as *Luftgitarre spielen*, and for German verbs with a more complex valency than nominative subject/accusative object, could enhance the coverage of the generator.

4.3 Possible Improvents of the Overall Design

Further possible improvements concern the overall design of the generator. The design choice to work without an explicit grammar and to encode linguistic knowledge in specialized predicates might have adverse consequences when it comes to extensibility. For example, as of now, the `dp/5` predicate in the `rules` module is defined by six Prolog rules, specialized to different kinds of semantic inputs, and quite some portions of code are duplicated. Future extensions, for example by additional arguments, risk causing a combinatorial explosion in the number of rules needed to define a predicate. If not an explicit grammar, then maybe a more rigid organization of the generating code could help to eliminate redundancies, ease extensibility and also increase efficiency by avoiding unsuccessful attempts to apply rules.

A final shortcoming of the interlingua used could be rectified without inventing or looking for a comletely different one. At present, BB1 undergoes the trouble of producing the full range of fully specified SRs for every sentence to be translated. GG1 then generates from each of them, often only to see identical syntactic outcomes in German and filter the duplicates. The procedure could be made more efficient by making the connection between BB1 and GG1 at a slightly lower level and adapting generation to the underspecified (hole semantics) formulae that BB1 works with internally. Differentiation into multiple forms would then be made in generation, and only if necessary. However, this would require a very considerate approach because hole semantics formulae are much less perspicuous than the fully specified logical formulae.

5 Conclusion

In this thesis I have shown how a first-order semantic representation language can be used as an interlingua in a very simple Machine Translation system which translates sentences of a fragment of English to German. After outlining some fundamental problems of generation from first-order formulae, I presented a pragmatic, yet reasonably systematic

approach, tailored to the semantic representation language used by Blackburn and Bos (2005).

Many features that are essential for more sophisticated Machine Translation systems are not incorporated in the system at all, such as components for resolving lexical ambiguities, transfer ambiguities, or anaphora (cf. Hutchins and Somers, 1992, chapter 5; p. 99–106). That it works without such components hinges in many ways on the very confined range of syntactic structures and lexical elements that the system deals with, inherited from the toy grammar that Blackburn and Bos use. The confined range of English sentences accepted as input is reflected in an equally confined range of semantic representations possible as output. This eased the initial development of the generation component significantly.

Slowly proceeding form very simple sentences to slightly more complex ones, it turned out that the structural richness of the first-order semantic representation language used becomes a difficult problem rather soon, since it hides logical meaning behind syntactic differences. Future work in combining Computational Semantics systems with Machine Translation should not use first-order logic but a more confined framework tailored to the needs of representing natural language semantics.

There is reason to believe that the general idea of the exercise at hand – to use a semantic representation language as an interlingua – could successfully be applied in more sophisticated MT systems. What makes semantic representation languages special as interlinguae is that they allow *inference*. This important property is not yet used in the present system, and this would be the next major step to take. Specifically, semantic representations should lend themselves well to *knowledge-based* MT (cf. Hutchins and Somers, 1992, p. 124 f.) that uses world knowledge and discourse knowledge built up over the course of translation to drive certain decisions in source text analysis and target text generation.

What can be said with certainty is that each of Machine Translation and Computational Semantics is partly about extracting some kind of information from natural language expressions. In one case, it is the information required for expressing the same content in another language, in the other case, it is the information required for performing inference. At least in my intutition these two kinds of information are not entirely different. Thus, beyond syntactic parsing, CS should be able to borrow many other techniques used in MT in order to achieve a better "understanding" of natural language expressions – techniques such as for resolving anaphora and ambiguities; or generation, in case a CS system is desired to communicate in natural language itself.

One preliminary for research at the interface of Computational Semantics and Machine Translation deserves especially careful consideration, namely common representations for CS and MT. The three criteria for SRLs proposed by Copestake et al. (1995, p. 18) – simplicity of structure, possibility of underspecification, and, of course, possiblity of inferencing – should be taken seriously. A very carefully designed and *stable* semantic representation language will probably prove to be very important, since changes that one component copes with well may pose fundamental difficulties to another – remember the difficulties discussed in section 2, of making grammars suitable for complete and efficient

parsing and generation alike.

For using Computational Semantics in Machine Translation, it will then be important to answer questions like the following precisely: How can analysis, transfer, and generation usefully interact with inferencing? What kind of models are useful for a translation system to build? What kind of information is useful for a translation system to query? The field of MT is so manifold that very different answers are imaginable.

References

Hiyan Alshawi, David Carter, Manny Rayner, and Björn Gambäck. Translation by quasi logical form transfer. In *Proceedings of the 29th Annual Meeting of the Association for Computational Linguistics*, pages 161–168, 1991.

Douglas E. Appelt. *Bidirectional Grammars and the Design of Natural Language Generation Systems*. Lawrence Erlbaum, 1989.

Patrick Blackburn and Johan Bos. Computational semantics. *Theoria*, 18(46):27–45, 2003.

Patrick Blackburn and Johan Bos. *Representation and Inference for Natural Language. A First Course in Computational Semantics*. CSLI, 2005.

Kai-Uwe Carstensen. Sprachtechnologie – ein Überblick. Web document, 2008. URL `http://www.cl.uzh.ch/CL/carstens/Materialien/SprachtechnologieCarstensen.pdf`.

Ann Copestake, Dan Flickinger, Rob Malouf, Susanne Riehemann, and Ivan Sag. Translation using Minimal Recursion Semantics. In *Proceedings of the Sixth International Conference on Theoretical and Methodological Issues in Machine Translation*, pages 15–32, July 1995.

Michael A. Covington. *Natural Language Processing for Prolog Programmers*. Prentice Hall, 1994.

H.-D. Ebbinghaus, J. Flum, and W. Thomas. *Mathematical Logic*. Undergraduate Texts in Mathematics. Springer-Verlag, second edition, 1994.

W. John Hutchins and Harold L. Somers. *An Introduction to Machine Translation*. Academic Press, 1992.

Martin Kay. Syntactic processing and functional sentence perspective. In *TINLAP '75: Proceedings of the 1975 Workshop on Theoretical Issues in Natural Language Processing*, pages 12–15. Association for Computational Linguistics, 1975.

Martin Kay. Chart generation. In *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics*, pages 200–204. Association for Computational Linguistics, 1996.

Jan Landsbergen. Montague grammar and machine translation. In P. Whitelock, M. M. Wood, H. L. Somers, R. Johnson, and P. Bennett, editors, *Linguistic Theory and Computer Applications*, pages 113–147. Academic Press, 1987.

David D. McDonald. Issues in the choice of a source for natural language generation. *Computational Linguistics*, 19(1):191–197, March 1993.

Richard Montague. The proper treatment of quantification in ordinary English. *Approaches to Natural Language*, 49:221–242, 1973.

John D. Phillips. Generation of Text from Logical Formulae. *Machine Translation*, 8 (4):209–235, 1993.

Ehud Reiter. Has a Consensus NL Generation Architecture Appeared, and is it Psycholinguistically Plausible? In *Proceedings of the Seventh International Workshop on Natural Language Generation*, pages 163–170, 1994.

Stuart M. Shieber. A uniform architecture for parsing and generation. In *Proceedings of the 12th International Conference on Computational Linguistics*, pages 614–619. Association for Computational Linguistics, 1988.

Stuart M. Shieber. The Problem of Logical-Form Equivalence. *Computational Linguistics*, 19(1):179–190, March 1993.

Stuart M. Shieber, Gertjan van Noord, Fernando C. N. Pereira, and Robert C. Moore. Semantic-head-driven generation. *Computational Linguistics*, 16(1):30–42, 1990.

Gertjaan van Noord. *An Overview of Head-driven Bottom-up Generation*, pages 141–165. Academic Press, 1990.

Graham Wilcock and Yuji Matsumoto. Head-driven generation with HPSG. In *Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics*, volume 2, pages 1393–1397. Association for Computational Linguistics, 1998.

A Prolog Code

A.1 translate.pl

```
/*************************************************************************

    File: translate.pl
    Copyright (C) 2008 Kilian Evang

    This file is part of GG1, version 1.0 (August 2008).

    GG1 is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.
```

```
    GG1 is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with BB1; if not, write to the Free Software Foundation, Inc.,
    59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

************************************************************************/

:- module(translate, [printTranslations/1,
                      translate/0,
                      translate/2]).

:- use_module('BB1/comsemPredicates', [memberList/2]).
:- use_module('BB1/holeSemantics', [holeSemantics/2]).
:- use_module('BB1/readLine', [readLine/1]).

:- use_module(rules, [s/2]).

/*======================================================================
   Top-level predicates for translating from English to German.
======================================================================*/

translate :-
   readLine(Sentence),
   % filter identical translations
   setof(Translation, translate(Sentence, Translation), Translations),
   printTranslations(Translations).

translate(Sentence, Translation) :-
   holeSemantics(Sentence, Sems),
   memberList(Sem, Sems),
   s(Sem, Translation).

/*======================================================================
   Converts syntactic trees to punctuated and capitalized strings.
======================================================================*/

printTranslations(Translations) :-
   printTran(Translations, 0).

printTran([], _) :-
   nl.

printTran([Translation|OtherTranslations], M) :-
   N is M + 1,
   nl,
   write(N),
   tab(1),
   leaves(Translation, Leaves),
   chunks(Leaves, Chunks),
   flatChunks(Chunks, [FirstFlatChunk|OtherFlatChunks]),
   capitalize(FirstFlatChunk, Capitalized),
   concat_atom([Capitalized|OtherFlatChunks], ', ', Sentence),
   atom_concat(Sentence, '.', Dotted),
   print(Dotted),
   printTran(OtherTranslations, N).

/*======================================================================
   A list of the occurences of atoms in the input formula, in the
   original order, with occurences of t eliminated, and the descendants
   of cp's enclosed as arguments of sub/N expressions, possibly nested.
======================================================================*/

leaves(t, []).

leaves(Formula, [Leaf]) :-
   Formula =.. [Leaf],
   \+ Leaf = t.

leaves(Formula, Leaves) :-
   Formula =.. [Cat|Children],
   Children = [_|_],
   \+ Cat = cp,
   listLeaves(Children, Leaves).

leaves(Clause, [Result]) :-
   Clause =.. [cp|Children],
   listLeaves(Children, Leaves),
   Result =.. [sub|Leaves].
```

```
listLeaves([],[]).

listLeaves([FirstFormula|OtherFormulae], Leaves) :-
   leaves(FirstFormula, FirstLeaves),
   listLeaves(OtherFormulae, OtherLeaves),
   append(FirstLeaves, OtherLeaves, Leaves).

/*========================================================================
   Input: A list of atoms, possibly recursively nested as arguments of
   predicates
   Output: A list of chunk/N expressions, possibly recursively nested
========================================================================*/

chunks(Leaves, Chunks) :-
   chunks(Leaves, [], [], Chunks).

chunks([], CurrentChunk, ChunksSoFar, Chunks) :-
   CurrentChunk =.. [_|_],
   ChunkExp =.. [chunk|CurrentChunk],
   append(ChunksSoFar, [ChunkExp], Chunks).

chunks([], [], ChunksSoFar, ChunksSoFar).

chunks([First|Rest], CurrentChunk, ChunksSoFar, Chunks) :-
   atom(First),
   append(CurrentChunk, [First], CurrentChunk1),
   chunks(Rest, CurrentChunk1, ChunksSoFar, Chunks).

chunks([First|Rest], CurrentChunk, ChunksSoFar, Chunks) :-
   First =.. [sub|Children],
   Chunk1Exp =.. [chunk|CurrentChunk],
   chunks(Children, Subchunks),
   Chunk2Exp =.. [chunk|Subchunks],
   append(ChunksSoFar, [Chunk1Exp, Chunk2Exp], ChunksSoFar1),
   chunks(Rest, [], ChunksSoFar1, Chunks).

/*========================================================================
   Input: A list of chunk/N expressions, possibly recursively nested
   Output: A list of lists of atoms
========================================================================*/

flatChunks(Chunks, FlatChunks) :-
   flatChunks(Chunks, [], FlatChunks).

flatChunks([], FlatChunksSoFar, FlatChunksSoFar).

flatChunks([FirstChunk|OtherChunks], FlatChunksSoFar, FlatChunks) :-
   FirstChunk =.. [chunk, FirstChild|OtherChildren],
   atom(FirstChild),
   concat_atom([FirstChild|OtherChildren], ' ', FlatChunk),
   append(FlatChunksSoFar, [FlatChunk], FlatChunksSoFar1),
   flatChunks(OtherChunks, FlatChunksSoFar1, FlatChunks).

flatChunks([FirstChunk|OtherChunks], FlatChunksSoFar, FlatChunks) :-
   FirstChunk =.. [chunk, FirstChild|OtherChildren],
   FirstChild =.. [chunk|_],
   flatChunks([FirstChild|OtherChildren], FlatSubchunks),
   append(FlatChunksSoFar, FlatSubchunks, FlatChunksSoFar1),
   flatChunks(OtherChunks, FlatChunksSoFar1, FlatChunks).

/*========================================================================
   Converts the first character of an atom to upper case
========================================================================*/

capitalize(Word, Capitalized) :-
   atom_chars(Word, [FirstLetter|OtherLetters]),
   upcase_atom(FirstLetter, Capital),
   atom_chars(Capitalized, [Capital|OtherLetters]).
```

A.2 rules.pl

```
/**************************************************************************

    File: rules.pl
    Copyright (C) 2008 Kilian Evang

    This file is part of GG1, version 1.0 (August 2008).

    GG1 is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
```

```
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.

    GG1 is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with BB1; if not, write to the Free Software Foundation, Inc.,
    59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

*************************************************************************/

:- module(rules, [s/2,
                  s/5]).

:- use_module('BB1/comsemPredicates', [memberList/2,
                                       selectFromList/3,
                                       substitute/4]).

:- use_module(lex).
:- use_module(util).

/*=======================================================================
   Top-level predicate. Arguments:
   1. A formula
   2. The generated sentence
=======================================================================*/

s(Formula, S) :-
   s(Formula, 0, false, svo, S).

/*=======================================================================
   Generation of sentences. Arguments:
   1. A formula
   2. How many quantifiers have already been "unscoped" into the formula
   3. Whether the sentence is to be negated
   4. Word order (svo or sov)
=======================================================================*/

% simple intransitive sentence
s(Formula, _, Neg, Order, S) :-
   Formula =.. [Pred, Subj],
   dp(Subj, _, nom, _, DP),
   vp_intrans(Pred, Neg, third, sing, Order, VP),
   S = s(DP, VP).

% simple transitive sentence
s(Formula, _, Neg, Order, S) :-
   Formula =.. [Pred, Subj, Obj],
   adjustTransNeg(Subj, Obj, NewSubj, NewObj),
   vp_trans(Pred, NewObj, Neg, third, sing, Order, VP),
   dp(NewSubj, _, nom, _, DP),
   S = s(DP, VP).

% sentence with "ist"
s(Formula, _, Neg, Order, S) :-
   Formula =.. [eq, PredN, Subj],
   setof((PredSex, VP, SubjSex, DP), (
      vp_be(PredN, PredSex, Neg, third, sing, Order, VP),
      dp(Subj, SubjSex, nom, _, DP)),
         Results),
   ((\+ memberList((Sex, VP, Sex, DP), Results),
     memberList((_, VP, _, DP), Results));
    memberList((Sex, VP, Sex, DP), Results)),
   S = s(DP, VP).

% universal quantification
s(all(X, imp(Restriction, NuclearScope)), Store, false, Order, S) :-
   NewStore is Store + 1,
   substitute(Y, X, Restriction, NewRestriction),
   substitute(q_all(Y, NewRestriction, NewStore), X, NuclearScope, NewFormula),
   s(NewFormula, NewStore, false, Order, S).

% negated universal quantification
s(all(X, imp(Restriction, NuclearScope)), Store, true, Order, S) :-
   NewStore is Store + 1,
   substitute(Y, X, Restriction, NewRestriction),
   substitute(q_notall(Y, NewRestriction, NewStore), X, NuclearScope,
              NewFormula),
   s(NewFormula, NewStore, false, Order, S).
```

```
% existential quantification
s(some(X, Scope), Store, false, Order, S) :-
   dissect(Scope, NuclearScope, Restriction),
   NewStore is Store + 1,
   substitute(Y, X, Restriction, NewRestriction),
   substitute(q_some(Y, NewRestriction, NewStore), X, NuclearScope, NewFormula),
   s(NewFormula, NewStore, false, Order, S).

% negated existential quantification
s(some(X, Scope), Store, true, Order, S) :-
   dissect(Scope, NuclearScope, Restriction),
   NewStore is Store + 1,
   substitute(Y, X, Restriction, NewRestriction),
   substitute(q_no(Y, NewRestriction, NewStore), X, NuclearScope, NewFormula),
   s(NewFormula, NewStore, false, Order, S).

% negation
s(not(Formula), Store, false, Order, S) :-
   s(Formula, Store, true, Order, S).

/*=========================================================================
   Dealing with negation
=========================================================================*/

% Object NPs not negatively quantified are unproblematic.
adjustTransNeg(Subj, Obj, Subj, Obj) :-
   \+ Obj =.. [q_no|_],
   \+ Obj =.. [q_notall|_].

% So are subject NPs that are not quantified.
adjustTransNeg(Subj, Obj, Subj, Obj) :-
   atom(Subj).

% When the subject NP is positively quantified and the object NP is negatively
% quantified, then if the latter quantification has wider scope the negation
% is moved to the subject NP.
adjustTransNeg(Subj, Obj, NewSubj, NewObj) :-
   nonvar(Subj),
   Subj =.. [SubjQPred, X, SubjScope, SubjStore],
   positive_qpred(SubjQPred),
   Obj =.. [ObjQPred, Y, ObjScope, ObjStore],
   negative_qpred(ObjQPred),
   ((ObjStore < SubjStore, inv_qpred(SubjQPred, InvSubjQPred),
                           NewSubj =.. [InvSubjQPred, X, SubjScope, SubjStore],
                           inv_qpred(ObjQPred, InvObjQPred),
                           NewObj =.. [InvObjQPred, Y, ObjScope, ObjStore]);
    (SubjStore < ObjStore, NewSubj = Subj,
                           NewObj = Obj)).

positive_qpred(q_some).
positive_qpred(q_all).

negative_qpred(q_no).
negative_qpred(q_notall).

inv_qpred(q_some, q_no).
inv_qpred(q_no, q_some).
inv_qpred(q_all, q_notall).
inv_qpred(q_notall, q_all).

/*=========================================================================
   Generation of determiner phrases. Arguments:
   1. Symbol of an individual or a quantified term
   2. Sex
   3. Case
   4. Gender
   5. The generated DP
=========================================================================*/

% name of an individual
dp(Indi, Sex, Case, Sex, DP) :-
   nonvar(Indi),
   n_proper(Indi, Sex, Case, N),
   DP = dp(N).

% trace
dp(t, _, _, _, dp(d(t))).

% universally quantified DP
dp(q_all(X, Restriction, _), Sex, Case, Gender, DP) :-
   d_univ(Case, sing, Gender, Infl, D),
   % Adjectives inflect strong iff the determiner is "not inflected":
   invertBool(Infl, Strong),
```

```
   np(X, Restriction, Sex, Case, sing, Gender, Strong, NP),
   DP = dp(D, NP).

% negative universally quantified DP
dp(q_notall(X, Restriction, _), Sex, Case, Gender, DP) :-
   d_univ_neg(Case, sing, Gender, Infl, D),
   invertBool(Infl, Strong),
   np(X, Restriction, Sex, Case, sing, Gender, Strong, NP),
   DP = dp(D, NP).

% existentially quantified DP
dp(q_some(X, Scope, _), Sex, Case, Gender, DP) :-
   d_indef(Case, sing, Gender, Infl, D),
   invertBool(Infl, Strong),
   np(X, Scope, Sex, Case, sing, Gender, Strong, NP),
   DP = dp(D, NP).

% negative existentially quantified DP
dp(q_no(X, Scope, _), Sex, Case, Gender, DP) :-
   d_indef_neg(Case, sing, Gender, Infl, D),
   invertBool(Infl, Strong),
   np(X, Scope, Sex, Case, sing, Gender, Strong, NP),
   DP = dp(D, NP).

/*=========================================================================
   Generation of noun phrases. Arguments:
   1. A variable
   2. Scope with propositions about the individual the variable stands for
   3. Case
   4. Number
   5. Gender
   6. Strong inflection
   7. The generated NP
=========================================================================*/

np(X, Scope, Sex, Case, Num, Gender, Strong, NP) :-
   flattenConjunction(Scope, FlatScope),
   sortProps(FlatScope, NounProps, AdjProps, RelProps),
   np(X, NounProps, AdjProps, RelProps, Sex, Case, Num, Gender, Strong, NP).

/*=========================================================================
   Generation of noun phrases - internal. Arguments:
   1. A variable
   2. List of simple formulae translating to nouns (length must be 1)
   3. List of simple formulae translating to adjectives
   4. List of complex formulae translating to relative clauses
   5. Case
   6. Number
   7. Gender
   8. Strong inflection
   9. The generated NP
=========================================================================*/

% NP with a relative clause
np(X, NounProps, AdjProps, RelProps, Sex, Case, Num, Gender, Strong, NP) :-
   selectFromList(RelProp, RelProps, OtherRelProps),
   np(X, NounProps, AdjProps, OtherRelProps, Sex, Case, Num, Gender, Strong,
      NP1),
   cp_rel(X, RelProp, Num, Gender, CP),
   NP = np(NP1, CP).

% NP with an AP
np(X, NounProps, AdjProps, [], Sex, Case, Num, Gender, Strong, NP) :-
   selectFromList(AdjProp, AdjProps, OtherAdjProps),
   np(X, NounProps, OtherAdjProps, [], Sex, Case, Num, Gender, Strong, NP1),
   ap(X, AdjProp, Case, Num, Gender, Strong, AP),
   NP = np(AP, NP1).

% simple NP
np(X, [NounProp], [], [], Sex, Case, Num, Gender, _, NP) :-
   NounProp =.. [Pred|X],
   n_common(Pred, Sex, Case, Num, Gender, N),
   NP = np(N).

/*=========================================================================
   Generation of relative clauses. Arguments:
   1. A variable
   2. A complex formula
   3. Number
   4. Gender
   5. The generated CP
=========================================================================*/
```

```
cp_rel(X, RelProp, Num, Gender, CP) :-
   % TODO analyze RelProp to allow for object relative clauses
   d_rel(nom, Num, Gender, _, D),
   substitute(t, X, RelProp, NewRelProp),
   s(NewRelProp, 0, false, sov, S),
   CP = cp(D, s(S)).

/*=========================================================================
   Generation of adjective phrases. Arguments:
   1. A variable
   2. A simple formula
   3. Case
   4. Number
   5. Gender
   6. Strong inflection
   7. The generated AP
=========================================================================*/

ap(X, AdjProp, Case, Num, Gender, Strong, AP) :-
   AdjProp =.. [AdjPred, X],
   a(AdjPred, Case, Num, Gender, Strong, A),
   AP = ap(A).

/*=========================================================================
   Generation of intransitive verb phrases. Arguments:
   1. Semantic symbol of the verb
   2. Whether the VP should be negated
   3. Person
   4. Number
   5. Word order (svo or sov)
   6. The generated VP
=========================================================================*/

vp_intrans(Pred, Neg, Person, Number, Order, VP) :-
   v(Pred, Person, Number, SimplexForm, ParticleForm),
   assembleVP(SimplexForm, ParticleForm, [], Neg, Order, VP).

/*=========================================================================
   Generation of transitive verb phrases. Arguments:
   1. Semantic symbol of the verb
   2. Symbol of an individual or a quantified term
   3. Whether the VP should be negated
   4. Person
   5. Number
   6. Word order (svo or sov)
   7. The generated VP
=========================================================================*/

vp_trans(Pred, Obj, Neg, Person, Number, Order, VP) :-
   v(Pred, Person, Number, SimplexForm, ParticleForm),
   dp(Obj, _, acc, _, DP),
   assembleVP(SimplexForm, ParticleForm, [DP], Neg, Order, VP).

/*=========================================================================
   Generation of verb phrases with "sein". Arguments:
   1. Semantic symbol of the verb
   2. Whether the VP should be negated
   3. Person
   4. Number
   5. Word order (svo or sov)
   6. The generated VP
=========================================================================*/

vp_be(PredN, Sex, Neg, Person, Number, Order, VP) :-
   v_be(Person, Number, SimplexForm, ParticleForm),
   dp(PredN, Sex, nom, _, DP),
   assembleVP(SimplexForm, ParticleForm, [DP], Neg, Order, VP).

/*=========================================================================
   Helper for the generation of verbal phrases
=========================================================================*/

assembleVP(SimplexForm, ParticleForm, DPs, Neg, Order, VP) :-
   ((ParticleForm = '', ParticleList = []);
    (\+ ParticleForm = '', ParticleList = [part(ParticleForm)])),
   SimplexList = [v(SimplexForm)],
   ((Neg = false, Nonheads = DPs);
    (Neg = true, append(DPs, [neg(nicht)], Nonheads))),
   ((Order = svo, append(SimplexList, Nonheads, Children1),
                  append(Children1, ParticleList, Children));
    (Order = sov, atom_concat(ParticleForm, SimplexForm, VForm),
                  append(Nonheads, [v(VForm)], Children))),
   VP =.. [vp|Children].
```

```
/*=========================================================================
   From a formula, extracts the proposition to use as the nucleus
   for the sentence to generate. For example, a suitable nucleus to
   extract from
   and(love(q_some(X, woman(X)), Y), boxer(Y))
   would be
   love(q_some(X, woman(X)), Y).
=========================================================================*/

dissect(Formula, Nucleus, Cloud) :-
   formula_conjuncts(Formula, Conjuncts),
   selectFromList(Nucleus, Conjuncts, OtherConjuncts),
   Nucleus =.. [Pred|_],
   \+ \+ isNuclear(Pred),
   formula_conjuncts(Cloud, OtherConjuncts).

/*=========================================================================
   Indicates for a given predicate symbol whether it can be the top
   predicate of a formula that is translated to a sentence or clause.
=========================================================================*/

isNuclear(all).

isNuclear(some).

isNuclear(not).

isNuclear(eq).

isNuclear(Pred) :-
   v(Pred, _, _, _, _).

/*=========================================================================
   Sorts formulae by what they are going to become: Nouns, adjectives, or
   relative clauses.
=========================================================================*/

sortProps([Prop|OtherProps], NounProps, AdjProps, RelProps) :-
   sortProps(OtherProps, OtherNounProps, OtherAdjProps, OtherRelProps),
   Prop =.. [Pred|_],
   ((\+ \+ isNuclear(Pred), NounProps = OtherNounProps,
                       AdjProps = OtherAdjProps,
                       RelProps = [Prop|OtherRelProps]);
    (\+ \+ a(Pred, _, _, _, _, _), NounProps = OtherNounProps,
                             AdjProps = [Prop|OtherAdjProps],
                             RelProps = OtherRelProps);
    (\+ \+ n_common(Pred, _, _, _, _, _), NounProps = [Prop|OtherNounProps],
                                    AdjProps = OtherAdjProps,
                                    RelProps = OtherRelProps)).

sortProps([], [], [], []).
```

A.3 lex.pl

```
/*************************************************************************

    File: lex.pl
    Copyright (C) 2008 Kilian Evang

    This file is part of GG1, version 1.0 (August 2008).

    GG1 is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.

    GG1 is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with BB1; if not, write to the Free Software Foundation, Inc.,
    59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

*************************************************************************/

:- module(lex, [n_common/6,
                n_proper/4,
```

```
                v_be/4,
                v/5,
                d_indef/5,
                d_indef_neg/5,
                d_univ/5,
                d_univ_neg/5,
                d_rel/5,
                a/6]).

:- use_module(stems).

/*=========================================================================
   Common nouns. Arguments:
   1. Semantic symbol
   2. Sex
   3. Case
   4. Number
   5. Gender
   6. Preterminal and terminal
=========================================================================*/

% sex-neutral nouns
n_common(animal, none, nom, sing, neut, n('Tier')).
n_common(animal, none, acc, sing, neut, n('Tier')).
n_common(beverage, none, nom, sing, neut, n('äGetrnk')).
n_common(beverage, none, acc, sing, neut, n('äGetrnk')).
n_common(building, none, nom, sing, neut, n('äGebude')).
n_common(building, none, acc, sing, neut, n('äGebude')).
n_common(cup, none, nom, sing, fem, n('Tasse')).
n_common(cup, none, acc, sing, fem, n('Tasse')).
n_common(burger, none, nom, sing, masc, n('Hamburger')).
n_common(burger, none, acc, sing, masc, n('Hamburger')).
n_common(car, none, nom, sing, neut, n('Auto')).
n_common(car, none, acc, sing, neut, n('Auto')).
n_common(chainsaw, none, nom, sing, fem, n('äKettensge')).
n_common(chainsaw, none, acc, sing, fem, n('äKettensge')).
n_common(drug, none, nom, sing, fem, n('Droge')).
n_common(drug, none, acc, sing, fem, n('Droge')).
n_common(episode, none, nom, sing, fem, n('Episode')).
n_common(episode, none, acc, sing, fem, n('Episode')).
n_common(fdshake, none, nom, sing, masc, n('üFnf-Dollar-Shake')).
n_common(fdshake, none, acc, sing, masc, n('üFnf-Dollar-Shake')).
n_common(footmassage, none, nom, sing, fem, n('ßFumassage')).
n_common(footmassage, none, acc, sing, fem, n('ßFumassage')).
n_common(gimp, none, nom, sing, masc, n('üKrppel')).
n_common(gimp, none, acc, sing, masc, n('üKrppel')).
n_common(glass, none, nom, sing, neut, n('Glas')).
n_common(glass, none, acc, sing, neut, n('Glas')).
n_common(gun, none, nom, sing, fem, n('Pistole')).
n_common(gun, none, acc, sing, fem, n('Pistole')).
n_common(hammer, none, nom, sing, masc, n('Hammer')).
n_common(hammer, none, acc, sing, masc, n('Hammer')).
n_common(hashbar, none, nom, sing, masc, n('Coffeeshop')).
n_common(hashbar, none, acc, sing, masc, n('Coffeeshop')).
n_common(person, none, nom, sing, fem, n('Person')).
n_common(person, none, acc, sing, fem, n('Person')).
n_common(joke, none, nom, sing, masc, n('Witz')).
n_common(joke, none, acc, sing, masc, n('Witz')).
n_common(needle, none, nom, sing, fem, n('Nadel')).
n_common(needle, none, acc, sing, fem, n('Nadel')).
n_common(piercing, none, nom, sing, fem, n('Durchbohrung')).
n_common(piercing, none, acc, sing, fem, n('Durchbohrung')).
n_common(plant, none, nom, sing, fem, n('Pflanze')).
n_common(plant, none, acc, sing, fem, n('Pflanze')).
n_common(qpwc, none, nom, sing, masc, n('Hamburger Royal')).
n_common(qpwc, none, acc, sing, masc, n('Hamburger Royal')).
n_common(radio, none, nom, sing, neut, n('Radio')).
n_common(radio, none, acc, sing, neut, n('Radio')).
n_common(restaurant, none, nom, sing, neut, n('Restaurant')).
n_common(restaurant, none, acc, sing, neut, n('Restaurant')).
n_common(suitcase, none, nom, sing, masc, n('Koffer')).
n_common(suitcase, none, acc, sing, masc, n('Koffer')).
n_common(shotgun, none, nom, sing, fem, n('Schrotflinte')).
n_common(shotgun, none, acc, sing, fem, n('Schrotflinte')).
n_common(sword, none, nom, sing, neut, n('Schwert')).
n_common(sword, none, acc, sing, neut, n('Schwert')).
n_common(vehicle, none, nom, sing, neut, n('Fahrzeug')).
n_common(vehicle, none, acc, sing, neut, n('Fahrzeug')).
n_common(weapon, none, nom, sing, fem, n('Waffe')).
n_common(weapon, none, acc, sing, fem, n('Waffe')).

% male/female pairs
n_common(boss, fem, nom, sing, fem, n('Chefin')).
```

```
n_common(boss, fem, acc, sing, fem, n('Chefin')).
n_common(boss, masc, nom, sing, masc, n('Chef')).
n_common(boss, masc, acc, sing, masc, n('Chef')).
n_common(boxer, fem, nom, sing, fem, n('Boxerin')).
n_common(boxer, fem, acc, sing, fem, n('Boxerin')).
n_common(boxer, masc, nom, sing, masc, n('Boxer')).
n_common(boxer, masc, acc, sing, masc, n('Boxer')).
n_common(criminal, fem, nom, sing, fem, n('Verbrecherin')).
n_common(criminal, fem, acc, sing, fem, n('Verbrecherin')).
n_common(criminal, masc, nom, sing, masc, n('Verbrecher')).
n_common(criminal, masc, acc, sing, masc, n('Verbrecher')).
n_common(customer, fem, nom, sing, fem, n('Kundin')).
n_common(customer, fem, acc, sing, fem, n('Kundin')).
n_common(customer, masc, nom, sing, masc, n('Kunde')).
n_common(customer, masc, acc, sing, masc, n('Kunde')).
n_common(owner, fem, nom, sing, fem, n('Besitzerin')).
n_common(owner, fem, acc, sing, fem, n('Besitzerin')).
n_common(owner, masc, nom, sing, masc, n('Besitzer')).
n_common(owner, masc, acc, sing, masc, n('Besitzer')).
n_common(robber, fem, nom, sing, fem, n('äRuberin')).
n_common(robber, fem, acc, sing, fem, n('äRuberin')).
n_common(robber, masc, nom, sing, masc, n('äRuber')).
n_common(robber, masc, acc, sing, masc, n('äRuber')).

% excklusively female
n_common(wife, fem, nom, sing, fem, n('Ehefrau')).
n_common(wife, fem, acc, sing, fem, n('Ehefrau')).
n_common(woman, fem, nom, sing, fem, n('Frau')).
n_common(woman, fem, acc, sing, fem, n('Frau')).

% exclusively male
n_common(husband, masc, nom, sing, masc, n('Ehemann')).
n_common(husband, masc, acc, sing, masc, n('Ehemann')).
n_common(man, masc, nom, sing, masc, n('Mann')).
n_common(man, masc, acc, sing, masc, n('Mann')).

/*==========================================================================
   Proper names. Arguments:
   1. Semantic symbol
   2. Sex/gender
   3. Case
   4. Preterminal and terminal
==========================================================================*/

n_proper(butch, masc, nom, n('Butch')).
n_proper(butch, masc, acc, n('Butch')).
n_proper(esmarelda, fem, nom, n('Esmarelda')).
n_proper(esmarelda, fem, acc, n('Esmarelda')).
n_proper(honey_bunny, fem, nom, n('Honey Bunny')).
n_proper(honey_bunny, fem, acc, n('Honey Bunny')).
n_proper(jimmy, masc, nom, n('Jimmy')).
n_proper(jimmy, masc, acc, n('Jimmy')).
n_proper(jody, fem, nom, n('Jody')).
n_proper(jody, fem, acc, n('Jody')).
n_proper(jules, masc, nom, n('Jules')).
n_proper(jules, masc, acc, n('Jules')).
n_proper(lance, masc, nom, n('Lance')).
n_proper(lance, masc, acc, n('Lance')).
n_proper(marsellus, masc, nom, n('Marsellus')).
n_proper(marsellus, masc, acc, n('Marsellus')).
n_proper(marvin, masc, nom, n('Marvin')).
n_proper(marvin, masc, acc, n('Marvin')).
n_proper(mia, fem, nom, n('Mia')).
n_proper(mia, fem, acc, n('Mia')).
n_proper(pumpkin, masc, nom, n('Pumpkin')).
n_proper(pumpkin, masc, acc, n('Pumpkin')).
n_proper(thewolf, masc, nom, n('Wolf')).
n_proper(thewolf, masc, acc, n('Wolf')).
n_proper(vincent, masc, nom, n('Vincent')).
n_proper(vincent, masc, acc, n('Vincent')).
n_proper(yolanda, fem, nom, n('Yolanda')).
n_proper(yolanda, fem, acc, n('Yolanda')).

/*==========================================================================
   The verb 'sein'. Arguments:
   1. Person
   2. Number
   3. Simplex form
   4. Particle (or empty atom)
==========================================================================*/

v_be(third, sing, 'ist', '').
```

```
/*=========================================================================
   Verbs. Arguments:
   1. Semantic symbol
   2. Person
   3. Number
   4. Simplex form
   5. Particle (or empty atom)
=========================================================================*/

% intransitive
v(collapse, third, sing, 'bricht', 'zusammen').
v(dance, third, sing, 'tanzt', '').
v(die, third, sing, 'stirbt', '').
v(growl, third, sing, 'knurrt', '').
v(smoke, third, sing, 'raucht', '').
v(snort, third, sing, 'prustet', '').
v(shriek, third, sing, 'kreischt', '').
v(walk, third, sing, 'geht', '').

% transitive
v(clean, third, sing, 'reinigt', '').
v(drink, third, sing, 'trinkt', '').
v(date, third, sing, 'trifft', '').
v(discard, third, sing, 'verwirft', '').
v(eat, third, sing, 'isst', '').
v(enjoy, third, sing, 'ßgeniet', '').
v(hate, third, sing, 'hasst', '').
v(have, third, sing, 'hat', '').
v(kill, third, sing, 'öttet', '').
v(know, third, sing, 'kennt', '').
v(like, third, sing, 'mag', '').
v(love, third, sing, 'liebt', '').
v(pickup, third, sing, 'holt', 'ab').
v(shoot, third, sing, 'ßschiet', 'ab').

/*=========================================================================
   Determiners. Arguments:
   1. Case
   2. Number
   3. Gender
   4. Strong inflection
   5. Preterminal and terminal
=========================================================================*/

d_indef(nom, sing, masc, false, d('ein')).
d_indef(acc, sing, masc, true, d('einen')).
d_indef(nom, sing, fem, true, d('eine')).
d_indef(acc, sing, fem, true, d('eine')).
d_indef(nom, sing, neut, false, d('ein')).
d_indef(acc, sing, neut, false, d('ein')).

d_indef_neg(nom, sing, masc, false, d('kein')).
d_indef_neg(acc, sing, masc, true, d('keinen')).
d_indef_neg(nom, sing, fem, true, d('keine')).
d_indef_neg(acc, sing, fem, true, d('keine')).
d_indef_neg(nom, sing, neut, false, d('kein')).
d_indef_neg(acc, sing, neut, false, d('kein')).

d_univ(nom, sing, masc, true, d('jeder')).
d_univ(acc, sing, masc, true, d('jeden')).
d_univ(nom, sing, fem, true, d('jede')).
d_univ(acc, sing, fem, true, d('jede')).
d_univ(nom, sing, neut, true, d('jedes')).
d_univ(acc, sing, neut, true, d('jedes')).

d_univ_neg(nom, sing, masc, true, d('nicht', 'jeder')).
d_univ_neg(acc, sing, masc, true, d('nicht', 'jeden')).
d_univ_neg(nom, sing, fem, true, d('nicht', 'jede')).
d_univ_neg(acc, sing, fem, true, d('nicht', 'jede')).
d_univ_neg(nom, sing, neut, true, d('nicht', 'jedes')).
d_univ_neg(acc, sing, neut, true, d('nicht', 'jedes')).

d_rel(nom, sing, masc, true, d('der')).
d_rel(acc, sing, masc, true, d('den')).
d_rel(nom, sing, fem, true, d('die')).
d_rel(acc, sing, fem, true, d('die')).
d_rel(nom, sing, neut, true, d('das')).
d_rel(acc, sing, neut, true, d('das')).

/*=========================================================================
   Adjectives. Arguments:
   1. Semantic symbol
   2. Case
```

```
   3. Number
   4. Gender
   5. Strong inflection
   6. Preterminal and terminal
========================================================================*/

a(Sem, nom, sing, masc, false, a(Syn)) :-
   a_stem(Sem, Stem),
   atom_concat(Stem, 'e', Syn).

a(Sem, acc, sing, masc, false, a(Syn)) :-
   a_stem(Sem, Stem),
   atom_concat(Stem, 'en', Syn).

a(Sem, nom, sing, masc, true, a(Syn)) :-
   a_stem(Sem, Stem),
   atom_concat(Stem, 'er', Syn).

a(Sem, acc, sing, masc, true, a(Syn)) :-
   a_stem(Sem, Stem),
   atom_concat(Stem, 'en', Syn).

a(Sem, nom, sing, fem, false, a(Syn)) :-
   a_stem(Sem, Stem),
   atom_concat(Stem, 'e', Syn).

a(Sem, acc, sing, fem, false, a(Syn)) :-
   a_stem(Sem, Stem),
   atom_concat(Stem, 'e', Syn).

a(Sem, nom, sing, fem, true, a(Syn)) :-
   a_stem(Sem, Stem),
   atom_concat(Stem, 'e', Syn).

a(Sem, acc, sing, fem, true, a(Syn)) :-
   a_stem(Sem, Stem),
   atom_concat(Stem, 'e', Syn).

a(Sem, nom, sing, neut, false, a(Syn)) :-
   a_stem(Sem, Stem),
   atom_concat(Stem, 'e', Syn).

a(Sem, acc, sing, neut, false, a(Syn)) :-
   a_stem(Sem, Stem),
   atom_concat(Stem, 'e', Syn).

a(Sem, nom, sing, neut, true, a(Syn)) :-
   a_stem(Sem, Stem),
   atom_concat(Stem, 'es', Syn).

a(Sem, acc, sing, neut, true, a(Syn)) :-
   a_stem(Sem, Stem),
   atom_concat(Stem, 'es', Syn).
```

A.4 stems.pl

```
/************************************************************************

    File: stems.pl
    Copyright (C) 2008 Kilian Evang

    This file is part of GG1, version 1.0 (August 2008).

    GG1 is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.

    GG1 is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with BB1; if not, write to the Free Software Foundation, Inc.,
    59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

************************************************************************/

:- module(stems, [a_stem/2]).
```

```
/*=========================================================================
   Adjectives
=========================================================================*/

a_stem(big, 'ßgro').
a_stem(blue, 'blau').
adj_stem(female, 'weiblich').
adj_stem(happy, 'öfrhlich').
adj_stem(male, 'ämnnlich').
adj_stem(married, 'verheiratet').
adj_stem(red, 'rot').
adj_stem(sad, 'traurig').
adj_stem(small, 'klein').
adj_stem(tall, 'ßgro').
```

A.5 util.pl

```
/*************************************************************************

    File: util.pl
    Copyright (C) 2008 Kilian Evang

    This file is part of GG1, version 1.0 (August 2008).

    GG1 is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.

    GG1 is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with BB1; if not, write to the Free Software Foundation, Inc.,
    59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

*************************************************************************/

:- module(util, [flattenConjunction/2,
                 formula_conjuncts/2,
                 invertBool/2]).

/*=========================================================================
   Flattening binary-branching conjunctions
=========================================================================*/

flattenConjunction(and(C1, C2), Flat) :-
   flattenConjunction(C1, Flat1),
   flattenConjunction(C2, Flat2),
   append(Flat1, Flat2, Flat).

flattenConjunction(Term, [Term]) :-
   \+ Term =.. [and|_].

/*=========================================================================
   Convert between formulas and lists of conjuncts, e.g.
   woman(X) <-> [woman(X)]
   and(woman(X), robber(X)) <-> [woman(X), robber(X)]
=========================================================================*/

formula_conjuncts(Formula, [Conjunct1, Conjunct2|OtherConjuncts]) :-
   Formula =.. [and, Conjunct1, Conjunct2|OtherConjuncts].

formula_conjuncts(Formula, [Conjunct]) :-
   nonvar(Formula),
   \+ Formula =.. [and|_],
   Conjunct = Formula.

formula_conjuncts(Formula, [Conjunct]) :-
   var(Formula),
   Formula = Conjunct.

/*=========================================================================
   Converts between the atoms true and false.
=========================================================================*/

invertBool(true, false).
```

```
invertBool(false, true).
```