ORAKEL: A Portable Natural Language Interface to Knowledge Bases

Philipp Cimiano, Peter Haase, Jörg Heizmann, Matthias Mantel

March 1, 2007

Chapter 1

Introduction

As the amount of information available globally on the Web and locally in intranets or databases keeps steadily growing, the necessity of mechanisms for effectively querying this information gains importance at the same pace. In fact, it seems crucial to provide end users with intuitive means of querying knowledge as they can not be expected to learn and use formal query languages such as SQL which are typically used by programmers. Different paradigms have been proposed in the past for querying information collections, among them *form fillin*, *query-by-example* or *menu-based approaches* (see [Shneiderman and Plaisant, 2005]), as well as natural language interfaces (NLIs), either relying on controlled language [Fuchs et al., 2006] or on more or less free language input [Popescu et al., 2003]. While the querying paradigm based on natural language is generally deemed to be the most intuitive from a usage point of view, it has also been shown to be the most difficult to realize effectively. The main reasons for this difficulty are that:

1. natural language understanding is indeed a very difficult task due to ambiguities arising at all levels of analysis: morphological, lexical, syntactic, semantic, and pragmatic (compare [Androutsopoulos et al., 1995, Copestake and Jones, 1989]),

2. a reasonably large grammar is required for the system to have an acceptable coverage,

3. the natural language interface needs to be accurate, and

4. the system should be adaptable to various domains without a significant effort.

With the wide availability of cell phones and PDAs, the importance of intuitive ways of interacting with electronic devices has grown even more. Natural language interfaces are an interesting option to interact with mobile devices due to their limited input and output functionality. Clearly, automatic speech

recognition is a crucial component towards leveraging the use of natural language interfaces. In this paper we are not concerned with speech recognition, but with the process of transforming a user's question into a formal query which can be answered with respect to an underlying knowledge or database. Nevertheless, it is worth emphasizing that speech recognition systems have nowadays reached a degree of maturity which makes it possible to apply them for interacting with phones or other mobile devices (see for example the recent SmartWeb project, which provides natural language access to the Semantic Web [Ankolekar et al., 2006]).

In the context of this paper, we define as *natural language interface* (NLI) any system accepting as input questions formulated in natural language and returning answers on the basis of a given knowledge base. It is important to emphasize that in our view a natural language interface goes strictly beyond the capabilities of keyword-based retrieval systems known from information retrieval research [Baeza-Yates and Ribeiro-Neto, 1999], which are not able to return precise answers to questions but only to return a set of relevant documents given a keyword-based query.

The ORAKEL natural language interface presented in this paper addresses all the above challenges, focusing particularly on minimizing the effort of adapting the system to a given domain. ORAKEL is an ontology-based natural language system in two senses. First, the ontology for a certain knowledge base is used to guide the lexicon construction process. On the one hand, parts of the lexicon are automatically generated from the underlying ontology. But most importantly, on the other hand, the ontology is at the core of the whole lexicon acquisition process in ORAKEL, which is performed by the user to adapt the system to some domain and particular knowledge base. Second, ORAKEL is ontology-based in the sense that it is a natural language interface which relies on deduction to answer a user's query. The ontology as a logical theory together with the facts stored in the knowledge base are thus exploited by the underlying inference engine to provide answer, even if it is not explicitly contained in the knowledge base but can be inferred from it. As ORAKEL relies on a well-defined deduction process to answer a query, an important requirement is that the user's question is translated into logical form, in particular into a query which can be evaluated by the underlying inference engine.

In general, the ontology model required by the system for the purposes of lexicon acquisition is rather simple, consisting of concepts, ordered hierarchically in terms of subsumption, as well as (binary) relations together with their corresponding restrictions on their domain and range (compare the ontology model described in [E. Bozsak et al., 2002] for a corresponding more formal definition). In practice, we will however rely on standard ontology models such as the ones provided by languages such as OWL [Bechhofer et al., 2004] or F-Logic [Kifer et al., 1995]. In fact, for the process of query answering, we will rely on the full expressive power of the logical languages used in the background.

The input to ORAKEL are factoid questions starting with so called *wh*-pronouns such as *who, what, where, which* etc., but also the expressions *'How many'* for counting and *'How'* followed by an adjective to ask for specific values

of an attribute as in *"How long is the Rhein?"*. Factoid in this context means that ORAKEL only provides ground facts as typically found in knowledge or data bases as answers, but no answers to *why*- or *how*-questions asking for explanations, the manner in which something happens or causes for some event.

The challenge for natural language interfaces is thus the domain-specific interpretation of the user's question in terms of relations and concepts defined in the schema or ontology of the knowledge base. Thus, parsers which create a generic logical form for a given input sentence will clearly not suffice for this purpose. The challenge is to construct a logical query consisting of domain-specific predicates which can be evaluated with respect to the knowledge base, returning the correct answer as a deduction process. Therefore, it is crucial that a natural language interface is adapted to every different knowledge base it is applied to.

In general, the problem of adapting natural language applications to some specific domain still remains largely unsolved. Different models for customization have been proposed in the natural language processing (NLP) literature. However, the feasibility of different customization approaches from a user point of view has been rarely investigated. While letting users engineer a complete grammar by hand might be a potential solution, it is for sure not feasible as it can neither be expected that general users have grammar engineering experience nor that they would be willing to make such an effort. Some systems support the user in defining linguistic rules, especially in the context of information extraction systems (compare [Cunningham et al., 1997]). In contrast, some researchers have examined supervised approaches in which training data is provided and the system learns domain-specific rules using inductive learning techniques [Thompson et al., 1997]. However, it seems still unclear whether providing training data, i.e. questions with their corresponding queries, is, from an end user point of view, a feasible way of customizing a natural language interface to a specific knowledge base. In general, the feasibility of different approaches from an end user point of view has been rarely investigated.

Finally, there are systems which support the user in lexicon acquisition by hiding the linguistic details behind some frontend. The well-known natural language interface TEAM [Grosz et al., 1987], for example, achieves the customization by asking domain experts questions and deriving the necessary linguistic knowledge from their answers. Rose et al. [Rose et al., 2005] have recently also presented an approach in which a NLP system is customized by users as a byproduct of annotating text segments. However, with the only exception of Rose et al. [Rose et al., 2005], none of the above work has examined the question whether typical users of the system are indeed able to successfully perform the customization.

In this paper, we explore a model of user-centered lexicon customization which merely requires very basic knowledge about subcategorization frames, but no background in computational or formal linguistics. Subcategorization frames are essentially linguistic argument structures, e.g. verbs with their arguments, nouns with their arguments, etc. As in TEAM, we also assume that a user with general expertise about computer systems will perform the cus-

tomization, i.e. we subscribe to the hypothesis mentioned in [Grosz et al., 1987]:

A major hypothesis underlying TEAM is that, if an NLI is constructed in a sufficiently well-principled manner, the information needed to adapt it to a new database and its corresponding domain can be acquired from users who have general expertise about computer systems and the particular database, but who do not possess any special knowledge about natural-language processing or the particular NLI.

In the ORAKEL system, the main task of the person in charge of customizing the system is to create a domain-specific lexicon mapping subcategorization frames to relations specified in the domain ontology. We present experimental evidence in form of a user study as well as in the form of a case study involving a real-world application to corroborate the claim that our model indeed allows non-NLP experts to create an appropriate domain lexicon efficiently and effectively. We show in particular that the results obtained with lexica customized by non-NLP experts do not substantially differ from the ones created by NLP experts. As the coverage of the lexicon has a direct impact on the overall linguistic coverage of the system, we propose a model in which the lexicon engineer can create the lexicon in an iterative process until a reasonable coverage is achieved. We also provide experimental evidence for the fact that such an iterative lexicon construction model is indeed promising. Furthermore, we also assess the coverage of our system, showing that with a few subcategorization frame types we can indeed yield a reasonable linguistic coverage. Before describing the details of ORAKEL, we first present an overview of the system in the next chapter.

Chapter 2

Overview of ORAKEL

In the ORAKEL system, we assume two underlying roles that users can play. On the one hand, we have *end users* of the system which interact with the system in query mode. On the other hand, domain experts or knowledge engineers which are familiar with the underlying knowledge base play the role of *lexicon engineers* which interact with the system in *lexicon acquisition mode*, creating domain-specific lexicons to adapt the system to a specific domain.

The end users ask questions which are semantically interpreted by the *Query Interpreter* (compare Figure 2.1). The Query Interpreter takes the question of the user, parses it and constructs a query in logical form (LF), formulated with respect to domain-specific predicates. This logical form is essentially a first order logic (FOL) representation enriched with query, count and arithmetic operators. The *Query Interpreter* component is discussed in detail in Chapter 3. The query in logical form is then translated by the *Query Converter* component into the target knowledge representation language of the knowledge base, in particular to its corresponding query language. The overall approach is thus independent from the specific target knowledge language and can accommodate any reasonably expressive knowledge representation language with a corresponding query language. Our system has been so far tested with the knowledge representation languages F-Logic [Kifer et al., 1995] with its query language as implemented by the Ontobroker system [Decker et al., 1999] and OWL [McGuiness and van Harmelen, 2004] with the query language SPARQL [Prud'hommeaux and Seaborne, 2006] as implemented by the KAON2 inference engine[1].

The conversion from the logical form to the target knowledge language is described declaratively by a Prolog program. The *Query Converter* component reads in this description and performs the appropriate transformation to the target query language. So far, we have provided the two implementations for F-Logic as well as OWL/SPARQL. However, our system architecture would indeed allow to port the system to any query language, in particular the RDF query

[1] http://kaon2.semanticweb.org/

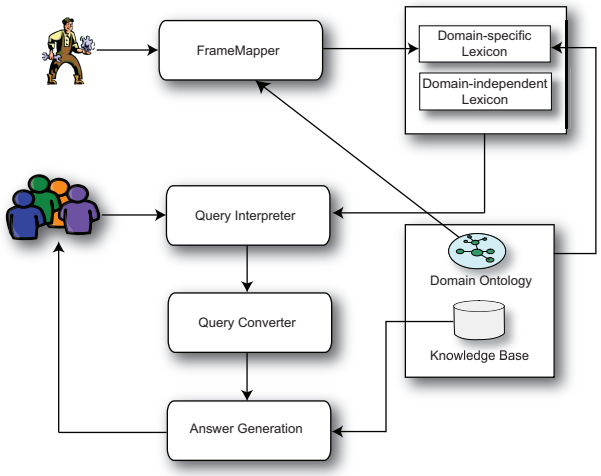


Figure 2.1: Overview of the ORAKEL system

languages described in [Haase et al., 2004] or plain SQL to access conventional relational databases. In fact, changing the target language requires a declarative description of the transformation as a Prolog program, but no further change to the underlying system. We describe the process of query translation in more detail in Section 3.4.

The answer generation component then evaluates the query with respect to the knowledge base and presents the answer to the user. Answering the query is thus a deduction process, i.e. the answer to a user's question are the bindings of the variables in the resulting query. Currently, the answer generation component only presents the extension of the query as returned by the inference engine. However, more sophisticated techniques for presenting the answer to the user by describing the answer intensionally or presenting the results graphically are possible. The way of displaying the results in general depends heavily on the application in question and will thus not be discussed further in this paper.

We have mentioned already in the introduction that a crucial question for natural language interfaces is how they can be adapted to a specific domain in order to interpret the user's question with respect to domain-specific predicates. In the model underlying ORAKEL, the lexicon engineer is in charge of creating a domain-specific lexicon thereby adapting ORAKEL to the domain in question. The lexicon engineer is essentially responsible for specifying how certain natural language expressions map to predicates in the knowledge base. For this purpose, we have designed an interface *FrameMapper* with access to the knowledge base, which supports the user in specifying by graphical means the mapping from language to relational predicates defined in the knowledge base. The result of the interaction of the knowledge engineer is a domain lexicon specific for the

application in question. The process of domain adaption is described in detail in Chapter 4, while the graphical user interface of FrameMapper is described in Chapter 5.

Besides the domain-specific lexicon, ORAKEL also relies on a general lexicon which specifies the semantics of closed-class words such as prepositions, determiners, question pronouns, numbers, etc. The semantics of these closed-class words are actually domain-independent and specified with respect to elementary or foundational categories as given by *foundational ontologies*. In our ORAKEL system, we rely on the foundational ontology DOLCE [Masolo et al., 2003], which provides fundamental categories such as *physical object*, *agentive physical object*, etc. as well as predicates and relations related to time and space. The latter ones are crucial for representing the semantics of spatial or temporal prepositions.

The general lexicon and the domain-specific lexicon created by the domain expert provide the only sources that ORAKEL needs to answer questions. Both type of lexica are in fact a lexicalized grammar which is used by ORAKEL for parsing but also for constructing the semantics of input questions. Thus, ORAKEL does not need any external grammar or other lexical resources[2]. As the general lexicon is given, the crucial bottleneck is thus the creation of the domain-specific lexicon. An appropriate domain-specific lexicon is crucial for interpreting the user's question with respect to domain-specific predicates. In this paper, our focus lies in particular on the adaption model and adaption mechanism of ORAKEL. Our aim is to show that, given very rudimentary knowledge about grammar and language, domain experts can indeed successfully adapt ORAKEL to different domains. We also show that an iterative approach in which the lexicon engineers modify the lexicon on the basis of failed questions until a reasonable coverage is achieved seems indeed reasonable.

We have carried out experiments on two different domains to corroborate our claim. On the one hand, we have carried out a user study with a small knowledge base containing facts about Germany. On the other hand, we have used a database containing metadata about research publications from British Telecom's – henceforth BT – digital library, which is orders of magnitude larger than the geography knowledge base. Our studies show that ORAKEL can indeed be successfully adapted to different domains in a reasonable amount of time, typically a few hours. The British Telecom case study was especially challenging as ORAKEL had to be modified to scale up to tens of thousands of facts contained in the BT database.

Further, we also address a few key issues mentioned already in the introduction. We also show that the precision of ORAKEL can compete with state-of-the-art systems as well as that the lexicon coverage is reasonably high. We also discuss additional features of the ORAKEL system, in particular its dis-

[2]The only two exceptions are lists of base forms for nouns and verbs with their corresponding inflected forms which are used by ORAKEL to generate tree families. This is discussed in more detail in Section 4. Further, WordNet is used to provide synonyms for verbs and nouns (compare Section 5). However, this possibility was not exploited in the experiments described in Chapter 6.

ambiguation and inferencing capabilities.

Chapter 3

Query Construction

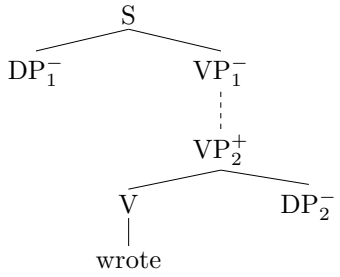
In this chapter, we describe how the logical query to the knowledge base is constructed on the basis of a user's question formulated in natural language. In the next section 3.1, we first describe the syntactic formalism underlying our system. In Section 3.2, we describe the parser of the system, which produces a syntactic analysis of the input sentence. Then, in Section 3.3 we describe how a query in our enriched first-order logic (FOL) language is constructed. Section 3.4 discusses how the FOL query can be translated into an appropriate target query language, e.g. into a F-Logic or SPARQL query in our case. In Section 3.5, we illustrate in detail all the different process steps on the basis of one example question. Finally, Section 3.6 describes how disambiguation is performed in our system, and Section 3.7 describes how inferencing capabilities are exploited in ORAKEL.

3.1 Logical Description Grammars (LDGs)

The underlying syntactic theory of our system is a formalism called Logical Description Grammars (LDG) (compare [Muskens, 2001]). LDG is inspired by Lexicalized Tree Adjoining Grammars (LTAGs) [Joshi and Schabes, 1997], which essentially are tree rewriting systems consisting of a finite set of trees associated with lexical items, so-called elementary trees (etrees). The two main operations in LTAG are substitution and adjoining. Substitution can be regarded as a local operation for the insertion of arguments. Adjoining typically folds one tree into another, thereby introducing modifiers or recursively embedding structures, such as clausal arguments. In general, Lexical Tree Adjoining Grammars exceed the computational power of context-free grammars and have been successfully used to model certain (syntactic) natural language phenomena (compare [Joshi and Schabes, 1997]).

The structures used in LDG are essentially (descriptions of) trees consisting of nodes labeled with syntactic information as depicted below. An important characteristic of these trees is that they encapsulate all syntactic/semantic ar-

guments of a word. The following tree for *wrote* for example explicitly indicates that it requires a subject (the author) at the DP_1 position as well as a direct object (the written document) at the DP_2 position. The fact that the line between VP_1 and VP_2 is dashed denotes that this dominance relation is not immediate, i.e. some other tree could slip in[1]. Typical trees which could slip in into this position are adverbs, e.g. *often*, or negation particles, e.g. *not*.



In essence, negatively marked nodes correspond to arguments which need to be inserted, while positively marked nodes denote variables to be inserted as an argument.

In the LDG formalism used in ORAKEL, there is only one operation, which consists in identifying positively with negatively marked nodes with each other within one or across trees. Hereby, two nodes can only be identified with each other if (i) they have complementary marks (negative/positive), (ii) they have the same syntactic category, (iii) their feature structures are compatible as well as (iv) syntactic dominance and surface order of words is respected. Feature structures in ORAKEL are in essence flat lists of attribute-value pairs. Two nodes can then only be made identical if they have the same value for a common attribute (see below the discussion of the features used in ORAKEL).

Substitution in LTAG corresponds in LDG to identifying the positively marked root node of some tree with the negatively marked leave node of some other tree, while effects as produced by the adjoining operation are achieved by the non-immediate dominance links as in the above tree for *wrote*. As mentioned above, these non-immediate dominance relations allow additional trees to slip in, which leads to comparable effects as those yielded by the adjoining operation in LTAG. In fact, both formalisms allow additional trees to be folded in, a mechanism used for instance for the insertion of modifiers, e.g. nominal, adjectival or adverbial modifiers.

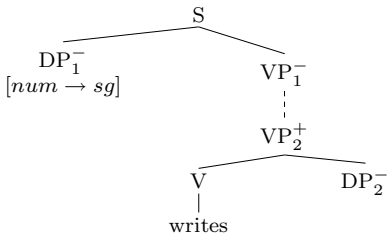
As noted above, the verb *write* requires a subject and an object. We say that *write subcategorizes* a subject and an object. It is therefore a *transitive* verb. However, there are not only transitive verbs, but also other types such as *intransitive verbs*, which subcategorize only a subject, *intransitive verbs with a prepositional complement*, *transitive verbs with a prepositional complement* as well as *ditransitive verbs* subcategorizing two objects. We call a verb together

[1]Here, DP standard for a *determiner phrase*, VP for a *verb phrase*, V for a verb and S for *sentence*.

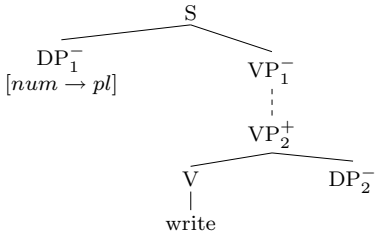
with a specification of which arguments it subcategorizes a *subcategorization frame*. Subcategorization frames are central in ORAKEL as they provide the basic structures which a lexicon engineer is supposed to map to domain-specific relations. Subcategorization frames give raise to another central notion: the one of *tree families*. Tree families encapsulate all the different ways in which a subcategorization frame can be expressed and thus capture generalizations of a given subcategorization frame type across words. For example, the tree family of a transitive verb such as *'write'* consists (at least) of the following trees:

- active:

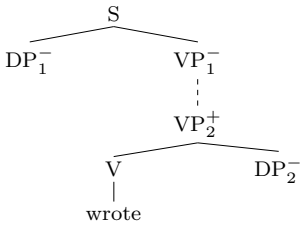
(1)



(2)

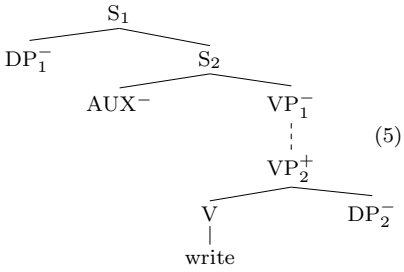


(3)

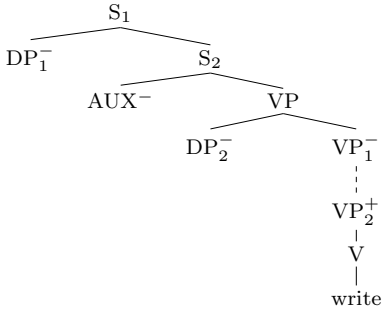


- auxiliary construction/object extraction:

(4)

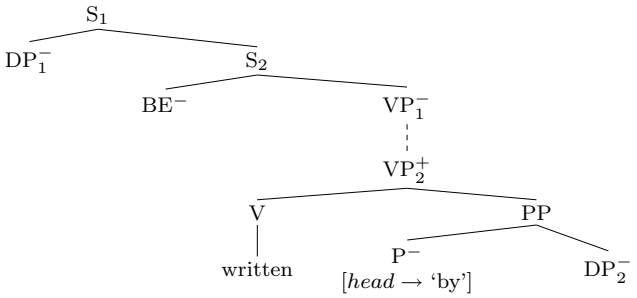


(5)



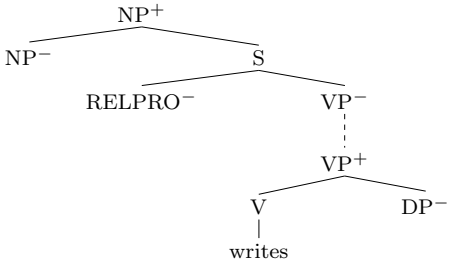
- passive:

(6)

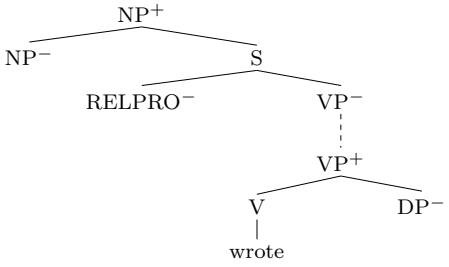


- relative clauses:

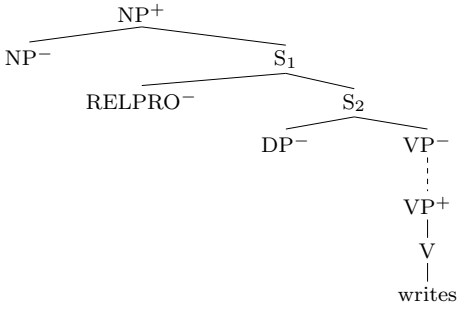
(7)



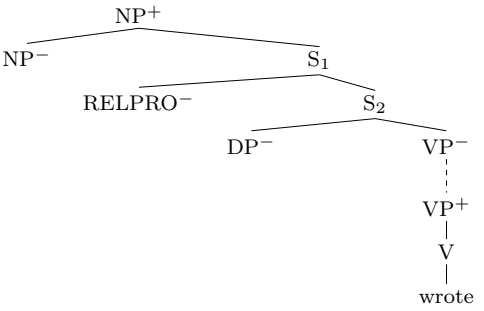
(8)



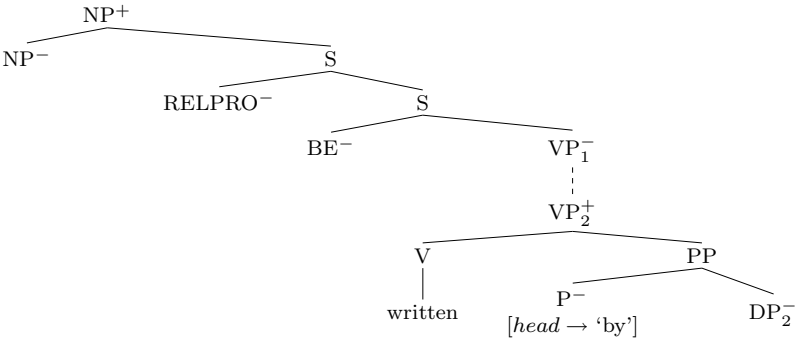
(9)



(10)



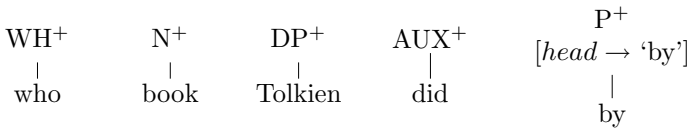
(11)



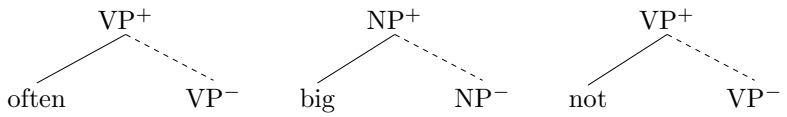
In the following, we give examples of the usage of the above trees, giving the number of the corresponding tree in parenthesis:

- Who writes/wrote a book? (1+3)
- Which authors write/wrote a book? (2+3)
- Who did not write a book? (4)
- Which book did Tolkien write? (5)
- Which book was written by Tolkien? (6)
- Who is the author who writes/wrote "The Lord of the Rings"? (7+8)
- Which is the book which Tolkien writes/wrote? (9+10)
- Which is the book which was written by Tolkien? (11)

For the sake of completeness, we also give the corresponding elementary trees for wh-pronouns such as *'who'*, common nouns, such as *'book'*, named entities such as *'Tolkien'*, auxiliaries such as *'did'* as well as prepositions such as *'by'*:



Modifiers such as adverbs, e.g. *'often'*, or adjectives, e.g. *'big'*, as well as *'not'* have all a similar syntactic structure as they are always optional and slip in between nodes connected by non-immediate dominance relations:



Currently, ORAKEL does not handle adverbial modification. The reason is that in the domains we have worked with so far it did not occur frequently enough to deserve consideration[2]. Further, allowing adverbial modification presupposes a rather complex modeling of temporal or spatial relations. On the contrary, adjective modification turned out to be very important and is thus handled by ORAKEL. The difficult issue certainly is to define the semantics of adjectives such as *'big'*, *'long'*, *'expensive'*, etc. in a reasonable way. Section 4 discusses in more detail the treatment of adjective modification.

Some nodes in the elementary trees shown so far contain feature-value pairs. As already mentioned, the feature-value pairs of different nodes need to be compatible in order to be identified with each other. In the ORAKEL system, we mainly rely on four features, i.e. *head*, *genus*, *function* and *type*. The *head* feature specifies the lexical head of some node. This is important in cases in which a verb subcategorizes a specific preposition. In this case we need to make sure that only the appropriate preposition is inserted at the P^- position (see for example the passive tree for *write* in (6) above). Further, the *genus* feature can be specified to be plural *(pl)* or singular *(sg)* and is crucial for establishing verb-subject agreement. The *function* feature can take the values *select* or *modify*. Taking the example of the adjective *'big'*, it has one elementary tree typed with *select*, which is used to select an appropriate attribute, i.e. the attribute *inhabitants* in a question like *"How big is Stuttgart?"*, as well as one elementary tree typed with *modify*, which is used as a modifier as in *"Which rivers flow through big cities?"*.

A further very important feature is *type*, which can be set either to *datatype property* or *object property*. This distinction is crucial when determining the semantics of expressions involving counting or numerical comparisons. To illustrate the importance of this distinction, consider two relations *inhabitants* and *flow_through*. When modeling the inhabitants of a city, region or country, we are normally not interested in the concrete individuals, but only in the total number of people living there. When modeling the rivers that flow through each city, we are normally not only interested in the number of rivers, but also in each of them as individuals. The relation *inhabitants* is thus typically modeled as a so called *datatype property*, i.e. as *inhabitants(city,integer)*, while the relation *flow_through* is normally modeled as a so called *object property*, i.e. as *flow_through(river,city)*. In spite of this essential difference in modeling, we can ask for numbers in similar ways regardless of the fact if they are modeled as

[2]This only holds with respect to our experience with natural language interfaces, but obviously not for natural language understanding in general.

datataype or *object properties*. We can ask questions related to the number of inhabitants of a city as follows:

- How many people live in Karlsruhe? (1a)
- In which city do the most people live? (1b)
- Which is the biggest city? (1c)
- Which city has more inhabitants than Karlsruhe? (1d)

Using similar constructs we can also ask for the number of rivers which flow through a city:

- How many rivers flow through Karlsruhe? (2a)
- Which city do the most rivers flow through? (2b)
- Which river flows through more cities than the Rhein? (2c)
- Which river flows through the most cities? (2d)

The crucial point is that language does not distinguish whether relations we talk about are modeled as *datatype* or *object properties*. The queries sent to a knowledge base, however, differ crucially depending on the fact whether a relation is modeled as a *datatype* or *object property* as the first type can be handled by standard numerical comparisons, while the second type involves counting operations. For illustration purposes, we show below the corresponding queries to a geographical knowledge base for the above example questions. In the remainder of this paper we will further make use of a generic query language exploiting standard first-order logic notation enriched with an additional query quantifier '?' binding the variables to be returned as answer. In what follows, lower case one-letter symbols denote variables while the other lower-case symbols denote constants:

- $?n\ inhabitants(karlsruhe, n)$ (1a)
- $?c\, \exists n_1\ inhabitants(c, n_1) \wedge city(c) \wedge \forall c', n_2\ (city(c') \wedge inhabitants(c', n_2) \rightarrow n_2 \leq n_1)$ (1b)
- $?c\, \exists n_1\ inhabitants(c, n_1) \wedge city(c) \wedge \forall c', n_2\ (city(c') \wedge inhabitants(c', n_2) \rightarrow n_2 \leq n_1)$ (1c)
- $?c\, \exists n_1\ inhabitants(c, n_1) \wedge city(c) \wedge \exists n_2 (inhabitants(karlsruhe, n_2) \wedge n_1 \geq n_2)$ (1d)
- $?n\ \exists r\ flow\_through(r, karlsruhe) \wedge count(\_, r, n)$ (2a)
- $?c\ \exists r, n_1\ city(c) \wedge flow\_through(r, c) \wedge count(c, r, n_1) \wedge \forall c', r', n_2 (flow\_through(r', c') \wedge count(c', r', n_2) \rightarrow n_1 \geq n_2)$ (2b)

- $?r\ \exists c, n_1\ river(r) \wedge flow\_through(r, c) \wedge count(r, c, n_1) \wedge$
  $\forall c', n_2(flow\_through(rhein, c') \wedge count(\_, c', n_2) \rightarrow n_1 > n_2)$ (2c)
- $?r\ \exists c, n_1\ river(r) \wedge flow\_through(r, c) \wedge count(r, c, n_1) \wedge$
  $\forall r', c', n_2(flow\_through(r', c') \wedge count(r', c', n_2) \rightarrow n_1 \geq n_2)$ (2d)

In the above queries, the count operator should be interpreted as follows: $count(a, b, n)$ is true if n is the number of elements b that a is related to in the way specified by the body of the query. In essence, the count operator thus groups all the b's according to the a's they stand in relation with and then counts the number of b's for each a. In case a is not specified, e.g. $count(\_, b, n)$, the b's are simply not grouped, thus yielding the absolute number of b's fulfilling the query.

The above example questions and corresponding queries illustrate two very important aspects. First of all, they show that the distinction between *datatype* and *object properties* is crucial to get the semantics right. In the case of a *datatype property*, we need to perform numerical comparisons, while in the case of an *object property*, we indeed need to count objects before performing a numerical comparison. As a consequence, we have in the general lexicon two different entries for the quantifiers *more, most, the most, more than, how many* as well as for numbers: one entry corresponding to an object property interpretation and one corresponding to a datatype property interpretation. Besides their different semantics, they differ in the value of the feature *type*, which is used to select the appropriate entry while parsing. This shows indeed that making sure that feature structures of identified nodes are compatible is also crucial for computing the correct semantic representation and not only to establish syntactic agreement, e.g. between the verb and its subject. We give the elementary trees for the above mentioned quantifiers and other standard quantifiers such as *'a'* and *'every'* in the Appendix.

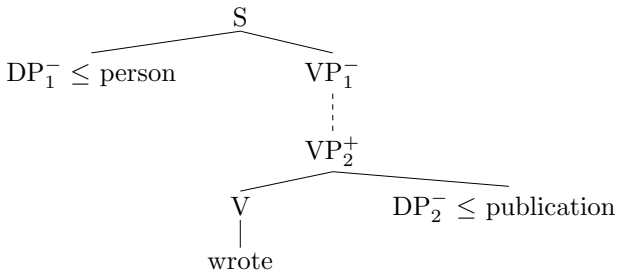
Second, the above examples also show that the way people ask for information rarely corresponds straightforwardly to the way information is modeled in an ontology. In particular, the examples show, on the one hand, that very different lexical and syntactic variants can be used to ask for the very same information. For example, to ask for the city with the most inhabitants we can either ask *"Which is the biggest city?"* - thus using a superlative, or *"Which city do the most people live in?"* - using the intransitive verb *'live'* with a prepositional complement introduced by the preposition *'in'*, or *"Which is the city with the most inhabitants?"* - using the preposition *'with'* followed by a noun phrase with head *'inhabitants'*, or *"Which city has the most inhabitants?"* using a similar construction involving the verb *'have'*.

On the other hand, similar constructions can be used to ask for information which is modeled differently in the ontology. For example, to ask for the number of inhabitants of a city, which is modeled as a *datatype property*, we can ask *"How many people live in Karlsruhe?"*, while when asking for the number of rivers which flow through a city, which is modeled through an *object property*, we can ask in the same way – modulo lexical differences due to the different relations involved – *"How many rivers flow through Karlsruhe?"*.

This exemplifies in fact that the correspondences between the way we talk about things and the way they are modeled in an ontology are far from straightforward. This shows why the problem of adapting a natural language interface is indeed a non-trivial task. As already mentioned, we support the customization of ORAKEL through a graphical user interface by which users can graphically specify how certain subcategorization frames map to relations (or joins of these) in the ontology. In the background, the system generates all the different syntactic variants as specified in the tree family of the corresponding subcategorization frame. The advantage of such an approach is that the semantics of each word needs to be specified exactly once by associating it with the corresponding subcategorization frames. Thus, all the generated trees from the corresponding tree family feature already the appropriate semantic representation.

We have illustrated above how the tree family for transitive verbs looks like. It is easy to imagine how the tree families for intransitive verbs with a prepositional complement, transitive verbs with a prepositional complement etc. look like. In ORAKEL, we also have tree families for adjectives as well as relational nouns. Relational nouns are those which subcategorize a prepositional complement, such as *mother (of)*, *brother (of)*, *capital (of)*. Typically, relational nouns can be used in a form in which the prepositional complement is existentially quantified, as in *"Which rivers flow through a capital?"*. Thus, for relational nouns, ORAKEL also generates variants in which the prepositional complement is not realized syntactically but existentially quantified (compare [Dekker, 1993] for a more deep and formal discussion of this issue). A description of the tree family of relational nouns featuring one prepositional complement is given in the Appendix.

In ORAKEL, we reuse a version of LDG augmented with selectional restrictions specified with respect to an ontology as described in [Cimiano and Reyle, 2003]. This ontological information is used to impose restrictions on the arguments of a predicate and thus for disambiguation. The tree for *wrote* would for example look as follows:



We will see later in Section 3.6 how these ontological restrictions are used for disambiguation.

In LDG, parsing boils down to identifying positively and negatively marked nodes with each other, respecting category information, feature values and surface order of words. The ORAKEL system implements a procedural version of LDG in which parsing proceeds as in typical LTAG parsers in two

stages. In fact, we implemented an Early-type bottom-up parser as described in [Schabes et al., 1988]. First, appropriate elementary trees for each word in the input are selected from the lexicon, and, second, these elementary trees are combined to yield a parse of the sentence (compare [Schabes et al., 1988]). In particular, ORAKEL relies on full parsing and does not make any use of partial parsing techniques. The parser used in the ORAKEL system is described in more detail in the next section.

3.2 Parsing

In ORAKEL, we have implemented a procedural version of the parsing mechanism inherent in the LDG approach. The parser basically identifies positively and negatively marked nodes respecting:

- the syntactic category of nodes,
- feature values,
- ontological constraints,
- surface word order, and
- syntactic dominance relations.

The parser is an Early-type bottom-up parser using top-down information as described in [Schabes et al., 1988] and [Schabes and Joshi, 1988]. It scans and reduces the input string from left to right, traversing the corresponding elementary trees in a top-down fashion. However, the parser can be called a bottom-up parser as it uses the words in the input string to guide the whole process.

The first component called is the so called *tree selector*, which gets as input the sentence and chooses appropriate elementary trees from the lexicon for each of the tokens in the input sentence. Assuming that there are at most m elementary trees for each token, the selector selects up to m^n forests with which the actual parser is called, where n is the length (in tokens) of the input sentence and m is the maximum number of different elementary trees for a token. This is a standard procedure for LTAG-like parsers (compare [Schabes et al., 1988]). In fact, this complexity can not be reduced in principle, but only by applying some sort of preprocessing to select the most promising trees, e.g. as done in supertagging [Bangalore and Joshi, 1999], and then only parsing the most promising forest.

Thus, the actual parser takes a forest of trees as input and first of all determines the elementary tree with its top node marked with r for *'root'* and starts parsing this elementary tree in a top-down fashion, recursively calling the parser for each child node. Algorithm 1 describes the parser in OO-style pseudocode. Before discussing the algorithm in detail, a few general comments are necessary. First of all, let us discuss the input to the algorithm. The parser gets as input the following parameters:

- **forest:** contains the input forest, i.e. a set of elementary trees corresponding to tokens from the input sentence,
- **ontology:** contains the background knowledge used by the parser, in our case only a hierarchy of concepts,
- **parseList:** contains all the valid parse trees for the current input sentence,
- **treeNum:** contains the number of the elementary tree in the forest which is currently being processed,
- **current node:** contains the node which is currently being parsed,
- **stack:** is a stack containing the next nodes to be processed, and
- **surface:** contains the part of the input string which still has to be processed; the surface will be iteratively reduced by removing tokens from the left corresponding to the parts of the input which have already been parsed.

The parser thus reads the input from left to right and constructs the parse tree in a top-down fashion, starting with the elementary tree marked as r for 'root'. The parser is called at the beginning with *surface* set to the input sentence, *treeNum* set to the elementary tree marked with r, and *node* to its root. The stack is also initially empty. Now we will discuss Algorithm 1 step by step.

The first *if*-condition contains the end condition for the recursion, i.e. the parser stops if the surface string and the stack are empty, that means, all the input has been processed, and there are no further nodes to process. The parse tree is added to the *parselist* after a final check that all the node identification operations indeed respect ontological constraints as well as that no cycles have been introduced. This is accomplished by the procedure *tree.consistent(ontology)*, which traverses the tree in a top-down fashion, verifying that the ontological constraints are compliant with the ontology as well as that there are no cycles.

If the end condition is not met, the parser checks if *node* is marked negatively. If this is the case, it will look for a corresponding positive node which is:

- dominated by the negative node in the same elementary tree (case 1), or
- non-immediately dominated by the (negative) node which has been already identified with the (positive) root of the current tree - the one with the number treeNum (case 2), or
- the root of an elementary tree to the left in case the current *node* is left from the path (or on the same path) from the root node to the lexical element of *tree* (this is verified by tree.Left.contains(node)) (case 3), or
- the root of an elementary tree to the right in case the current *node* is right from the path (or on the same path) from the root to the lexical element in *tree* (this is verified by tree.Right.contains(node)) (case 4)

Algorithm 1 Parsing Algorithm

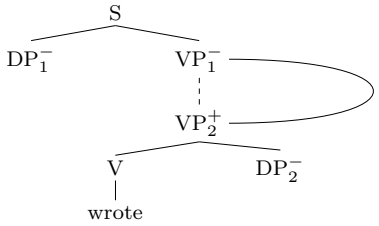
```
parse(forest, ontology, parselist, treeNum, node, stack, surface)
{
  if (surface.length == 0 && stack.isEmpty()) {
    // forest has been parsed succesfully
    for (int i = 0; i < forest.ElementaryTrees.size(); i++) {
      tree = (ElementaryTree) forest.ElementaryTrees.get(i);
      if (tree.getRoot().getMark().equals("r") && tree.consistent(ontology))
        parselist.add(forest);
    }
  }
  else {
    tree = (ElementaryTree) forest.ElementaryTrees.get(treeNum);
    if (negative(node)) {
      // case 1
      foreach child in node.Dominated() {
        tmpStack = stack.cloneStack();
        if (child.match(node)) {
          newForest = forest.cloneAndIdentify(node,child,next,tmpStack);
          parse(newForest,ontology,parselist,treeNum,next,tmpStack,surface);
        }
      }
      if (tree.getRoot().Reverse) {
      // case 2
        foreach child in tree.Root.Reverse.Dominated() {
          tmpStack = stack.cloneStack();
          if (child.match(node)) {
            newForest = forest.cloneAndIdentify(node,plusNode,next,tmpStack);
            parse(newForest,ontology,parselist,treeNum,next,tmpStack,surface);
          }
        }
      }
      if (tree.Left.contains(node) || tree.Path.contains(node)) {
      // case 3
        for (int j = treeNum-1; j >= 0; j=j-1) {
          tmpStack = stack.cloneStack();
          tmpTree = (ElementaryTree) forest.ElementaryTrees.get(j);
          if (candidate.getRoot().match(node)) {
            newForest = forest.cloneAndIdentify(node,candidate.getRoot(),next,tmpStack;
            parse(newForest,ontology,parselist,j,next,tmpStack,surface);
          }
        }
      }
      if (tree.Right.contains(node) || tree.Path.contains(node)) {
      // case 4
        for (int j = treeNum+1; j < forest.ElementaryTrees.size(); j=j+1) {
          tmpStack = stack.cloneStack();
          candidate = (ElementaryTree) forest.ElementaryTrees.get(j);
          if (candidate.getRoot().match(node)) {
            newForest = forest.cloneAndIdentify(node,candidate.getRoot(),next,tmpStack);
            parse(newForest,ontology,parselist,j,next,tmpStack,surface);
          }
        }
      }
    }
  }
```

Algorithm 2 Parsing Algorithm (Cont'd)

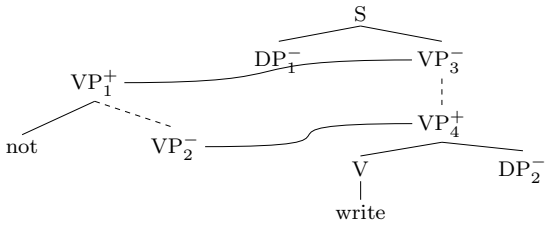
```
   else
   // node is not negative
   {
     if (hasChildren(node)) {
       tmpStack = stack.cloneStack();
       newForest = forest.clone(tmpStack,node,next);
       foreach child in next.Dominated()
       {
         tmpStack.push(next,treeNum);
       }
       stackObject = tmpStack.pop();
       parse(ontology,newForest,parselist,stackObject.getPosition(),
             stackObject.getNode(),tmpStack,surface);
     }
     else {
       // node is a leaf
       newSurface = surface - node.getLexem();
       if (newSurface == null) return;
       else {
         if (!stack.isEmpty()) {
           tmpStack = stack.cloneStack()
           newForest = forest.clone(tmpStack);
           stackObject = tmpStack.pop();
           parse(ontology,newForest,parselist,stackObject.getPosition(),
           stackObject.getNode(),tmpStack,newSurface);
         }
       }
     }
   }
 }
}
```

Hereby, for some elementary tree t, $t.Left$ contains the nodes to the left from the path from the root to the lexical elements, $t.Right$ contains the elements to the right and $t.Path$ contains the nodes from the root node to the lexical element.

Case 1 corresponds to the situation in which a positively marked node and a negatively marked node non-immediately dominating it are identified, e.g. as in:



Thus, in this case no modifiers can slip in between the two VP nodes. Of course, once the parser backtracks other identifications are possible as for example in case 2, which accounts for the situation in which a modifier tree slips in:



It is interesting to note that in the above configuration, we first process the elementary tree for *'not'* and no identification is performed for VP_2 as none of the cases 1-4 holds. When processing the tree for *'write'*, we would identify the negatively marked VP_3-node with the positively marked VP_1 root node of the elementary tree for *'not'*. The *next* node would then be set to the VP_1 node of the *'not'* elementary tree. The *root.getReverse()* pointer of the *'not'* elementary tree would then have been set to the negatively marked VP_3 node in the *'write'* tree. Thus, according to case 2 of our algorithm, the negatively marked VP_2 node in the elementary tree for *'not'* would be identified with the positively marked VP_4 node dominated by the *reverse pointer* of the VP_1 root node of the *'not'* elementary tree, i.e. the VP_3 node. In general, it is important to mention that every time two nodes are identified, a *reverse pointer* is maintained from the positively marked to the negatively marked node.

Cases 3 and 4 are relatively straightforward; a negatively marked node to the left (right) of the path from the root node to the lexical element is identified with the positive root node of an elementary tree to the left (right) of the tree being processed. This accounts for standard substitution of arguments.

In all cases, the algorithm checks if the nodes match, i.e. if the syntactic category, feature values and ontological constraints are respected by the procedure

match. If this is the case, a new forest and a new stack are created. The new forest *newForest* is basically a copy of the old forest in which both nodes have been identified. The new stack *newStack* is a copy of the old stack in which all elements have been replaced by their copies. The parser is then recursively called with the *newForest* and *newStack* as well as with the new node to be processed, i.e. *next*. The procedure *forest.cloneAndIdentify(node,candidate,next,tmpStack)* creates a copy of the current forest in which the nodes *candidate* and *node* have been identified. Further, *next* is set to the copy of the formerly positively marked node *candidate*, which is then processed next. The nodes in the temporal stack *tmpStack* are updated accordingly so that they refer correctly to the nodes in the cloned forest. The reason for creating copies of the stack and the forest is to allow backtracking and thus a depth-first search in the space of possible parse trees. Creating local copies of the data structures could also be avoided by undoing identifications after the recursion has finished. However, from an implementation point of view it is definitely easier to create local copies instead of undoing the effects after the recursion.

In case the node to be processed is not negative, there are two further cases to consider:

- the node has children
- the node has no children, i.e. it is a lexical element.

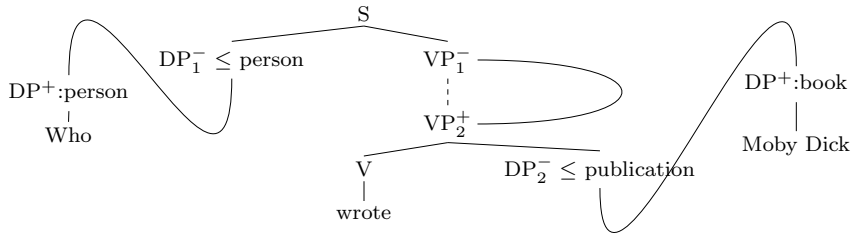
In case it has children, the forest and the stack are cloned and each child is added to the new stack. The parser is then recursively called with the new forest and stack as well as with the oldest element added to the stack as node to be processed. In case it has no children[3], the node is a lexical element which is used to reduce the surface from the left-hand side (this is done by the $-$ operator). In case the lexical element can not be left-substracted from the surface, then $newSurface$ will be undefined and the parser will stop as the parse tree currently in process of construction can not be extended to a full parse tree anymore due the fact that it will not correctly cover the input sentence. In case the $newSurface$ string is well-defined, the parser retrieves the next node to be processed from the stack and enters a new recursion with this new node.

It is important to emphasize that only the negative nodes are processed and identified with appropriate positive nodes. This is indeed sufficient as the result of the parsing is essentially a bijection between negative and positive nodes, such that it makes no difference if we only process the negative or the positive nodes. However, it makes the formulation and implementation of the algorithm much easier. The fact that at the end negatively and positively marked nodes need to be bijectively identified also allows to implement a simple heuristic which allows to dramatically reduce the parser's runtime. This heuristic consists in counting the number of positively and negatively marked nodes for each category before calling the parser. In case the numbers differ, no bijection is possible and thus no parse will be found.

[3]Note that positive nodes are assumed not to be leaves and thus never considered at this stage.

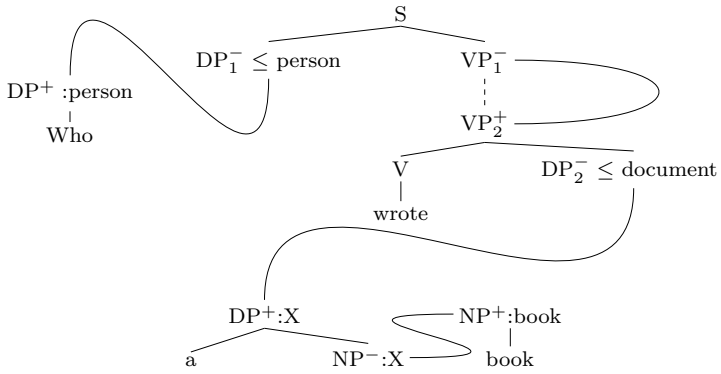
A further interesting aspect to emphasize are the ontology-based consistency checks. Indeed, while some ontology checks can be performed locally by the *node.match(candidate)* procedure, other checks have to be performed globally by the *tree.consistent(onto)* procedure. We illustrate the distinction between local and global consistency checks with two examples.

Example 1 (local consistency check):



The fulfillment of the ontological restrictions on the subject ($\leq$ person) and object ($\leq$ publication) positions of *write* can indeed be verified locally in the above configuration as the positively marked DP-nodes are typed. Hereby, $\leq$ denotes the conceptual subsumption operator which mirrors the class hierarchy of the underlying ontology. In this particular example, the restrictions are fulfilled assuming that $person \leq person$ and $book \leq publication$. This differs from the configuration in the following example:

Example 2 (global consistency check):



In this configuration, the DP root node of the elementary tree for *'a'* is not typed and only gets its type after the NP^- node has been identified with the root of the elementary tree for *'book'*. Thus, the verification of ontological restrictions can not be performed locally, which shows the necessity of applying a consistency check after the elementary trees have been assembled to a complete parse tree. This is exactly what is achieved by the *tree.consistency(onto)* procedure.

This concludes the description of the parsing algorithm. The process of semantic construction is described in the next section, whereas section 3.5 gives a complete example for the parser and the semantic construction component.

3.3 Semantics Construction

ORAKEL implements a compositional semantics approach to construct the logical formula corresponding to the input question. Compositional means here that the query to the database or knowledge base - i.e. the formal semantics of the input sentence - is recursively computed on the basis of the meaning of every single word in the input sentence as well as the way the words are connected. Thus, the logical query representing a question is constructed en par with the syntactic analysis of the question. Such an approach requires some sort of syntactic processing grouping words to larger syntactic units and ordering them as trees to guide the recursive computation. This is accomplished by the parser described in the previous section.

The semantics of a sentence is then the semantics of the top node of the elementary tree marked as root and is specified by a FOL-like formula which is translated in a subsequent step to a formula in the target query language via a Prolog conversion program.

The semantic construction proceeds en par with the syntactic construction in a traditional compositional manner (compare [Montague, 1974]). Thereby, each node specifies how its meaning is constructed on the basis of the meaning of its children using the lambda calculus. In ORAKEL we use an extended version of the lambda calculus implemented in Prolog by Blackburn and Bos ([Blackburn and Bos, 2005]). The main extensions are:

- the addition of a new non-standard compositional operator $\oplus$
- the introduction of marked referents to be substituted by $\oplus$

We will argue below why the introduction of this operator is necessary. Our extension of the lambda calculus implementation of Blackburn and Bos as well as the Prolog program for translating the FOL output into F-Logic or SPARQL can be downloaded at `http://www.cimiano.de/orakel`.

A compositional semantics construction approach as implemented by ORAKEL requires relatively rich lexical resources specifying the logical meaning of each word. This is exactly where our user-centered model for lexicon customization fills a gap as the rich semantic lexicon is generated in the background as a byproduct of the interaction of the user with the system's lexicon acquisition frontend, called *FrameMapper* (see Section 5). Details about the semantics of each word remain completely hidden to the user. Indirectly, the user is thus generating a grammar as well as associating logical meanings to words without even being aware of it. We will discuss this process in detail in Sections 4 and 5. Each lambda expression in our approach is typed. We only consider the two primitive types e for entities/individuals and t for truth value. While e thus denotes all the individuals in the domain of discourse, t denotes the truth values *true* and *false*. Types are recursively defined as follows:

Definition 1 (Types)

- t_1 *is a type if* $t_1 \in \{e, t\}$,

- $< t_1, t_2 >$ *is a type if* t_1 *and* t_2 *are types, and*
- *nothing else is a type.*

Hereby, $< t_1, t_2 >$ stands for the type of a function taking an expression of type t_1 as argument and returning an expression of type t_2.

As a short illustrating example, imagine a user asking the question: *"Which river passes through Berlin?"* to a knowledge base containing facts about German geography. The meaning of the diverse lexico-syntactic units in the input can be expressed in functional lambda notation together with their corresponding type after the slash as follows:

Which river	$\lambda P\ ?x\ (river(x) \wedge P(x))\ /\ << e, t >, t >$
passes through	$\lambda x\, \lambda y\ flow\_through(x, y)\ /\ < e, < e, t >>$
Berlin	$\lambda Q\ Q(Berlin)\ /\ << e, t >, t >$

So the semantic representation of *'passes through'* is of type $/ < e, < e, t >>$ and thus expects two arguments of type e, i.e. individuals, to be inserted into the appropriate relation *flow_through*. The expression *'which river'* is of type $<< e, t >, t >$ and thus expects some property P of type $< e, t >$, i.e. a function from individuals to truth values, which x, a river, needs to fulfill. *'Berlin'* is also of type $<< e, t >, t >$ and thus also expects some predicate Q into which it can be inserted as an argument.

Given the simplified syntactic structure together with instructions how the semantic expressions are applied to each other in Figure 3.1, and evaluating the tree in a standard bottom-up fashion, we would first carry out the functional application

$$\lambda u\ (\lambda Q\ Q(Berlin))((\lambda x\, \lambda y\ flow\_through(x, y))(u)),$$

yielding as semantic representation of the VP node

$$\lambda u\ flow\_through(u, Berlin)/ < e, t >$$

in which the argument *Berlin* has been correctly inserted. To yield the final semantic representation of the top sentence node S, we would carry out the functional application

$$(\lambda P\, ?x\, (river(x) \wedge P(x)))(\lambda u\ flow\_through(u, Berlin)),$$

resulting in the final logical query:

$$?x\ (river(x) \wedge flow\_through(x, Berlin))/t$$

In some cases, we will however omit the type information for the sake of presentation. In our approach to compositional semantics, we rely on the following three operations:

- β-conversion/reduction for functional application,

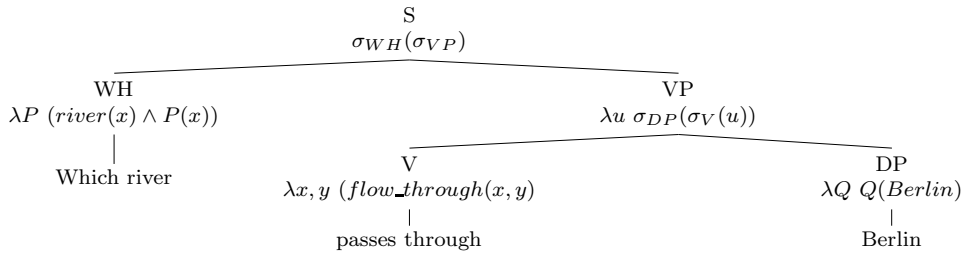


Figure 3.1: Syntactic analysis with semantic representations for each word specified according to the λ-calculus and instructions how to combine the different representations with each other.

- α-conversion for renaming bound variables, and
- substitution of marked variables by the $\oplus$ operator.

Here follow the formal definitions:

Definition 2 (β-conversion/reduction (functional application))
Let s_1 be an expression of the form $\lambda x\ s'_1$, where x occurs somewhere in s'_1 and s_1 has the type $< t_2, t_1 >$. Let furthermore s_2 be an expression of type t_2, then the result of applying s_2 to s_1 is $s_1(s_2) := s'_1[x/s_2]$, where $s'_1[x/s_2]$ means that x is substituted in s'_1 by s_2. Further, $s_1(s_2)$ has the type t_1.

As an illustrating example, imagine $s_1 = \lambda Q\ Q(Berlin)\ /\ << e, t >, t >$ and $s_2 = \lambda x\ ?w\ flow\_through(w, x)\ /\ < e, t >$, then $s_1(s_2) = ?w\ flow\_through(w, Berlin)\ /\ t$.

For the β-conversion to be well-defined s_1 has to be free for the variables in s_2. This means that no variable occurring in s_2 should be bound by an operator in s_1, whereby operator refers to any quantifier. Thus, β-reduction may make it necessary to rename bound variables in t_1. This is achieved with α-conversion:

Definition 3 (α-conversion)
Given a term s where the variable x occurs bound by some operator, i.e. some quantifier, then we call $\alpha(s) := s[x/y]$, where y is some variable not occurring in s, an α-variant of s. The process of renaming a bound variable is called α-conversion.

As an example for α-conversion, imagine we have $s_1 = \lambda P\ \forall w\ (river(w) \wedge P(w))\ /\ << e, t >, t >$ and $s_2 = \lambda x\ ?w\ flow(w, x)\ /\ < e, t >$, then before performing β-conversion we should rename s_1 to $\lambda P\ \forall x(river(x) \wedge P(x))\ /\ << e, t >, t >$, thus yielding $(\alpha(s_1))(s_2) = \forall x\ (river(x) \wedge ?w\ flow(w, x))\ /\ t$. Now we come to marked substitution:

Definition 4 (Marked substitution)
If s_1 is an expression with an occurrence of a marked variable x, i.e. x' occurs in s_1, the result of marked substitution is $s_1 \oplus s_2 := (\lambda x'\ s_1)(s_2)$.

As an example of the marked substitution operator, imagine the expression $s_1 = flow\_through(x', y)$ where the variable x is marked, and the expression $s_2 = Rhein$. As a result of $s_1 \oplus s_2$ we would get $flow\_through(Rhein, y)$. We will see below in Section 3.5 when discussing an illustrating example why this operation is useful.

3.4 Query Conversion

In order to increase its flexibility, ORAKEL has been designed to be, on the one hand, domain independent and, on the other hand, independent of the specific knowledge representation and query language used in the background. Domain independence is achieved by separating the general and domain lexica as is typically done for transportable NLIs (compare [Grosz et al., 1987]). The latter one needs to be handcrafted by a domain expert. The independence of the target logical language is achieved by introducing a First-Order-Logic (FOL) language enriched with additional predicates for quantifiers as well as query and numerical operators, which is produced by our semantic analysis component. The question *"Which city do the most rivers flow through?"* is for example represented as follows in our FOL-like query language:

$?c\ \exists r, n_1\ city(c) \wedge flow\_through(r, c) \wedge count(c, r, n_1) \wedge$
$\forall c', r', n_2\ (flow\_through(r', c') \wedge count(c', r', n_2) \rightarrow n_1 \geq n_2)$

Queries in this FOL-like language can then be translated to any logical language by a translation component. Hereby, the translation is specified declaratively in Prolog and is thus exchangeable[4]. The Prolog conversion programs essentially specify recursively how the operators of the query language ($?$, $\exists$, $\wedge$, $\rightarrow$, $count(...)$). The above query is for example translated into F-Logic as follows:

$\forall C \leftarrow \exists R, N_1\ C : City \wedge R[flow\_through \rightarrow C] \wedge count(C, N, N_1) \wedge$
$\forall C', R', N_2\ (R'[flow\_through \rightarrow C'] \wedge count(C', R', N_2) \rightarrow geq(N_1, N_2))$

While all the queries specified in our FOL-like query language can be translated into F-Logic, this is not the case for the SPARQL language. Currently, the SPARQL language essentially supports only conjunctive queries such that the above query would not be translatable to SPARQL.

A direct translation to some target formalism as performed in [Cimiano, 2003] is also possible, but clearly such an approach is not as flexi-

[4]The Prolog code for the conversion into F-Logic and SPARQL can be found at `http://www.cimiano.de/orakel`.

ble as the one pursued within ORAKEL. Currently, our system supports two formalisms used in the Semantic Web, the Web Ontology Language (OWL)[5] with the query language SPARQL[6] as well as F-Logic as ontology language together with its corresponding query language [Kifer et al., 1995]. The ontologies essentially provide the schema for the knowledge base and thus the concepts and relations relevant for the domain in question. This system design allows to port our system to any domain and any (reasonably expressive) logical formalism with a query language. The only requirement on the language is that it provides extra-logical predicates for counting and for numerical comparisons.[7]

3.5 An illustrating example

In what follows, we discuss in detail the parse tree and semantic construction for the question: *"Which rivers flow through more cities than the Rhein?"*. The elementary trees for this question are given in Figure 3.5. We will assume that the *tree selector* has selected already these elementary trees and calls the parser with the 8th elementary tree corresponding to the question mark, which is marked with an r, as well as the S_3 node as first node to process. In what follows, we discuss the construction of the parse tree for the above example step by step:

1st call:

treeNum = 9
node = S_3
stack = [];
identified = []
surface = Which rivers flow through more cities than the Rhein ?

As S_3 is not positively marked, the parser simply adds the children of S_3, i.e. S_4 and '?', to the stack, popping again S_4 and recursively calling the parser to process this node.

2nd call:

treeNum = 9
node = S_4^-
stack = [?]
identified = []
surface = Which rivers flow through more cities than the Rhein ?

[5] http://www.w3.org/TR/owl-ref/
[6] http://www.w3.org/TR/rdf-sparql-query/
[7] This is currently not met by SPARQL, thus leading to a reduced expressivity in the target language.

As S_4 is negatively marked, the parser then attempts to find a correspondingly positively marked node. As S_4 is to the left of the path from the root node to the lexical element '?', the parser searches for an appropriate root node of an elementary tree to the left. The only possible candidate is S_1. So, both nodes are identified and the parser is recursively called with S_1:

3rd call:

treeNum = 3
node = S_1
stack = [?]
identified = [(S_4,S_1)]
surface = Which rivers flow through more cities than the Rhein ?

As S_1 is now not marked anymore, the algorithm simply adds its children - DP_2 and S_2 - to the stack, processing first the negatively marked DP_2, i.e.

4th call:

treeNum = 3
node = DP_2^-
stack = [S_2,?]
identified = [(S_4,S_1)]
surface = Which rivers flow through more cities than the Rhein ?

The only candidate to be identified with DP_2 is DP_1, and the parser is called recursively with DP_1 in the

5th call:

treeNum = 1
node = DP_1
stack = [S_2,?]
identified = [(S_4,S_1),(DP_2,DP_1)]
surface = Which rivers flow through more cities than the Rhein ?

As DP_1 is not negatively marked, both children WH and NP_1 are put on the stack and the parser recursively called with the WH node:

6th call:

treeNum = 1
node = WH
stack = [NP_1^-,S_2,?]

identified = [(S_4,S_1),(DP_2,DP_1)]
surface = Which rivers flow through more cities than the Rhein ?

Node WH has one child, which is further processed and, as it is a lexical node, the surface string is left-reduced by *'which'* in the

7th call:

treeNum = 1
node = *which*
stack = [NP_1^-,S_2,?]
identified = [(S_4,S_1),(DP_2,DP_1)]
surface = rivers flow through more cities than the Rhein ?

The parser then returns to process the first element of the stack, i.e. np_1:

8th call:

treeNum = 1
node = NP_1^-
stack = [S_2,?]
identified = [(S_4,S_1),(DP_2,DP_1)]
surface = rivers flow through more cities than the Rhein ?

As NP_1 is on the right of the path from the root to the lexical element of the elementary tree 1, the parser searches for a corresponding root of an elementary tree on the right, thus identifying NP_1 with NP_2 and calling the parser recursively for NP_2. Here we abbreviate a little bit as it is clear what happens, i.e. the surface string is left-reduced with *'rivers'* in the 9th call, thus leading to a recursive call in which the element S_2 of the stack is processed, i.e.

10th call:

treeNum = 3
node = S_2
stack = [?]
identified = [(S_4,S_1),(DP_2,DP_1),(NP_1,NP_2)]
surface = flow through more cities than the Rhein ?

Then the parser is recursively called with the child VP_1^-:

11th call:

treeNum = 3
node = VP_1^-

stack = [?]
identified = [(S_4,S_1),(DP_2,DP_1),(NP_1,NP_2)]
surface = flow through more cities than the Rhein ?

As VP_1 is negatively marked and the parser finds one dominated node which matches it, i.e. VP_2, both nodes are identified and the parser continues processing VP_2:

12th call:

treeNum = 3
node = VP_2
stack = [?]
identified = [(S_4,S_1),(DP_2,DP_1),(NP_1,NP_2),(VP_1,VP_2)]
surface = flow through more cities than the Rhein ?

The rest of the parsing proceeds analogously, so we abbreviate it:

- **14th call**: the surface string is left reduced by *'flow'*
- **16th call**: P_1 is identified with P_2
- **18th call**: the surface string is left reduced by *'through'*
- **19th call**: DP_3 is identified with DP_4
- **21st call**: the surface string is left reduced by *'more'*
- **22nd call**: NP_3 is identified with NP_4
- **24th call**: the surface string is left reduced by *'cities'*
- **25th call**: P_3 is identified with P_4
- **27th call**: the surface string is reduced by *'than'*
- **28th call**: DP_5 is identified with DP_6
- **30th call**: the surface string is left reduced by *'the Rhein'*

The 31st call then finally processes the question mark:

31st call:

treeNum = 8
node = ?
stack = []
identified = [(S_4,S_1),(DP_2,DP_1),(NP_1,NP_2),(VP_1,VP_2),(P_1,P_2),(DP_3,DP_4),(NP_3,NP_4),
(P_3,P_4),(DP_5,DP_6)]
surface = ?

In this 31st call, the surface string becomes thus empty. Finally, in the 32nd call, as the surface string and the stack are empty, the forest is added to the parselist. As a result, the parser yields the node identifications as depicted in Figure 3.5. Concerning the semantics, we get the following:

$$
\begin{aligned}
\sigma_{dp_1} = \quad & \sigma_{wh}(\sigma_{np_1}) = \sigma_{wh}(\sigma_{np_2}) \\
= \quad & (\lambda P\ \lambda Q\ ?w\ (P(w) \wedge Q(w)))(\lambda x\ river(x)) \\
= \quad & \lambda Q\ ?w\ (river(w) \wedge Q(w)) \\
\sigma_{dp_4} = \quad & (\sigma_{more}(\sigma_{dp_5}))(\sigma_{np_3}) = (\sigma_{more}(\sigma_{dp_6}))(\sigma_{np_4}) \\
& ((\lambda R\ \lambda T\ \lambda S\ \exists v_1\, \exists v_2\, \exists v_3\, \exists v_4\ (T(v_1) \wedge R(\lambda h\, (S(v_1) \oplus h)) \wedge (S(v_2) \oplus c) \wedge \\
& T(v_2) \wedge count(c, v_2, v_3) \wedge count(\_, v_1, v_4) \wedge v_1 > v_4))(\lambda P\ P(rhein)))(\lambda x\ city(x)) \\
= \quad & (\lambda T\ \lambda S\ \exists v_1\, \exists v_2\, \exists v_3\, \exists v_4\, (T(v_1) \wedge \lambda P\ P(rhein)(\lambda h\, (S(v_1) \oplus h)) \wedge (S(v_2) \oplus c) \wedge \\
& T(v_2) \wedge count(c, v_2, v_3) \wedge count(\_, v_1, v_4) \wedge v_3 > v_4))(\lambda x\ city(x)) \\
= \quad & \lambda S\ \exists v_1\, \exists v_2\, \exists v_3\, \exists v_4\, (city(v_1) \wedge (S(v_1) \oplus rhein) \wedge (S(v_2) \oplus c) \wedge \\
& city(v_2) \wedge count(c, v_2, v_3) \wedge count(\_, v_1, v_4) \wedge v_3 > v_4) \\
\sigma_{vp_2} = \quad & \lambda u\ \sigma_{pp_1}(\sigma_v(u)) = \lambda u\ \sigma_{dp_3}(\sigma_v(u)) = \lambda u\ \sigma_{dp_4}(\sigma_v(u)) \\
= \quad & \lambda u\ (\lambda S\ \exists v_1\, \exists v_2\, \exists v_3\, \exists v_4\, (city(v_1) \wedge (S(v_1) \oplus rhein) \wedge (S(v_2) \oplus c) \wedge \\
& city(v_2) \wedge count(c, v_2, v_3) \wedge count(\_, v_1, v_4) \wedge v_3 > v_4))(\lambda y\ (flow(u, y)) \\
= \quad & \lambda u\ \exists v_1\, \exists v_2\, \exists v_3\, \exists v_4\, (city(v_1) \wedge flow(rhein, v_1) \wedge \\
& flow(u, v_2) \wedge city(v_2) \wedge count(u, v_2, v_3) \wedge count(\_, v_1, v_4) \wedge v_3 > v_4) \\
\sigma_{s_1} = \quad & \sigma_{s_2}(\sigma_{dp_2}) = \sigma_{vp_1}(\sigma_{dp_1}) = \sigma_{vp_2}(\sigma_{dp_1}) \\
= \quad & ?w\ (river(w) \wedge \exists v_1\, \exists v_2\, \exists v_3\, \exists v_4\, (city(v_1) \wedge flow(rhein, v_1) \wedge flow(w, v_2) \wedge \\
& city(v_2) \wedge count(w, v_2, v_3) \wedge count(\_, v_1, v_4) \wedge v_3 > v_4))
\end{aligned}
$$

The final semantics of the sentence is thus in FOL:

$$?w\ (river(w) \wedge \exists v_1\, \exists v_2\, \exists v_3\, \exists v_4\, (city(v_1) \wedge flow\_through(rhein, v_1) \wedge flow\_through(w, v_2) \wedge city(v_2) \wedge count(w, v_2, v_3) \wedge count(\_, v_1, v_4) \wedge v_1 > v_4)$$

Notice that the above first-order query is not a generic logical form but one which makes already use of domain-specific predicates defined in the underlying ontology. Our F-Logic translator then finally translates this into the F-Logic query:

$$\forall W \leftarrow W : river \wedge \exists V1\, \exists V2\, \exists V3\, \exists V4\, (V1 : city \wedge rhein[flow\_through \rightarrow V1] \wedge W[flow\_through \rightarrow V2] \wedge V_2 : city \wedge count(W, V2, V3) \wedge count(\_, V1, V4) \wedge greater(V3, V4)).$$

The above example also illustrates the function of the $\oplus$ operator. In fact, when comparing entities involved in certain relations using operators such as *'more than'*, *'most'*, etc., we need to compare the extensions of the relation for one particular entity compared to all the other entities. For example, when asking *"Which river flows through more cities than the Rhein?"* we need to compare the number of cities n the Rhein flows through, i.e.

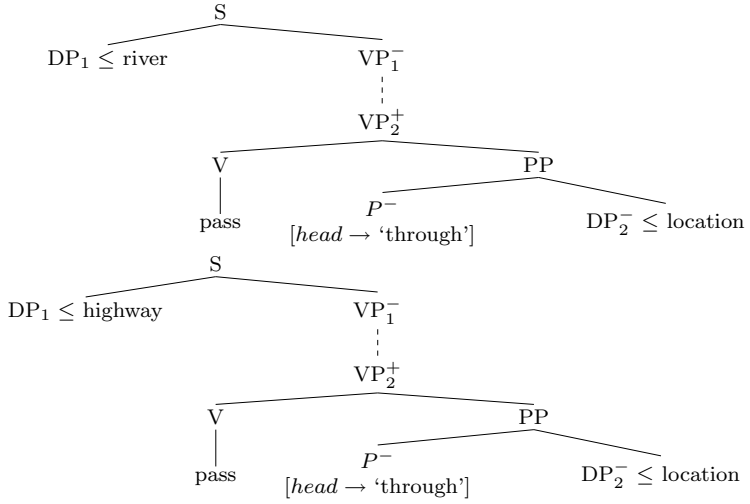
$flow\_through(rhein, c_1) \wedge count(\_, c_1, n)$ to the number of cities $f(r)$ that other rivers r flow through, i.e. $flow\_through(r, c_2) \wedge count(r, c_2, f(r))$ and return those rivers r for which $f(r) > n$. To accomplish this, we need to know the c_1's which stand in the relation $flow\_through(rhein, c_1)$ as well as the c_2's which stand in the relation $flow\_through(r, c_2)$ with any r. A standard compositional semantics interpretation would merely insert the subject *'which river'* into the domain position of the $flow\_through$ relation. However, when composing the semantics of *'more than'*, we have no access to the domain of the relation $flow\_through$, which is filled only when the subject of the sentence is processed according to a standard compositional process. However, we need to fill the domain position with *'rhein'*. The solution here is thus our $\oplus$ operator which is able to unify the domain of the *flow_through* relation with *'rhein'*, thus providing the information necessary for the above described comparisons. This relies thus on the assumption that the domain position of the *flow_through* relation is marked for substitution. In fact, in our approach, we always assume that the position of the relation associated with the subject is marked for substitution. However, note that our use of the $\oplus$ operator does unfortunately not provide a general solution as it relies on the fact that the subject will never be a comparative expression. A comparative expression at subject position could be found in an assertion such as *"Most rivers flow through Karlsruhe."* However, if we restrict ourselves to questions, it seems hard to imagine a question with a comparative quantifier at the subject position, so that our solution seems satisfactory for a natural language interface, but not for the task of natural language understanding in general.

So far, we have discussed step-by-step the parsing, semantic construction as well the conversion for the example in question. This concludes thus our detailed description of ORAKEL's core components. In what follows, we briefly describe two further features of our natural interface, i.e. its disambiguation and inferencing capabilities.

3.6 Disambiguation

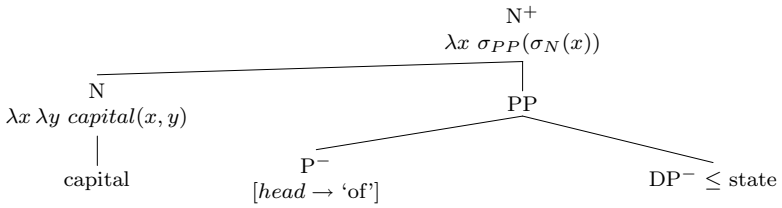
Concerning disambiguation, imagine a question like: *"Which river passes through Berlin?"*. In this case, the verb *'passes through'* is ambiguous with respect to a geographical knowledge base as it can denote the relation *flow_through* relating rivers to locations they flow through as well as a relation *located_at_highway* relating highways to cities they pass by. The way disambiguation is achieved in ORAKEL is by ensuring that arguments match the appropriate selectional restrictions of a predicate. Thus, while *'pass'* is lexically ambiguous with respect to these different relations, after verifying the selectional restrictions, only the interpretation in the sense of rivers flowing through a city remains available as *'rivers'* fulfills the ontological restriction of the relation *flow_through*, but not the one of the *located_at_highway* relation. This disambiguation is achieved already at parsing time as only components which fulfill each other's ontological restrictions are combined. In general, this disam-

biguation strategy is similar to the one implemented in the *ontological semantics* framework of Nirenburg and Raskin [**?**]. For illustration, consider the following elementary trees for *'flow through'* and *'passes through'*, which illustrate how selectional restrictions are specified:



As another example, consider the question: *"What is the capital of Berlin?"* In fact, in our lexicon *'Berlin'* is ambiguous between the state of Berlin and the city of Berlin. In particular, the lexicon contains the following entries:

The elementary tree for *'capital'* looks as follows and clearly selects for a *state* at the DP$^-$ position:



Thus, the above elementary tree clearly selects for the state interpretation of *'Berlin'*, thus disambiguating its meaning correctly.

3.7 Inferencing

Concerning inferencing, rules or other axioms can be added to the ontology in order to infer further implicit relationships. We could have a rule stating that the *location* relation is transitive, i.e.

$$\forall x, z\ location(x, z) \leftarrow \exists y\ location(x, y) \wedge location(y, z).$$

Further, we could have a rule stating that a river x flows through a location z if x flows through y and y is located in z:

$$\forall x, z\ flow\_through(x, z) \leftarrow \exists y\ flow\_through(x, y) \wedge location(y, z).$$

Now assuming that in the knowledge base we have specified for all rivers the cities they flow through as well as the location of cities to states and states to countries, a question like *"Which countries does the Donau flow through?"*, translated into $?w\ country(w) \wedge flow\_through(donau, w))$, would return the appropriate answer relying on the above rules.

Another rule currently available in the system is one which defines *borders* as a symmetric relation, i.e.

$$\forall x, y\ borders(x, y) \leftarrow borders(y, x).$$

By modeling the above rule, we only need to specify the bordering relation in one direction, while the other will be inferred. By this move, we halve the size of the border-relation in the knowledge base.

In general, as ORAKEL relies on an inference engine and an axiomatized ontology in the background, it can exploit the full inferencing capabilities of the logical language and the inference engine used in the background. In general, the inference capabilities of ORAKEL thus highly depend on the expressiveness and inferences supported by the logical language used. In our settings, while OWL provides us essentially with subsumption reasoning, F-Logic allows us to use the expressive power of Horn rules to express relationships as the above.

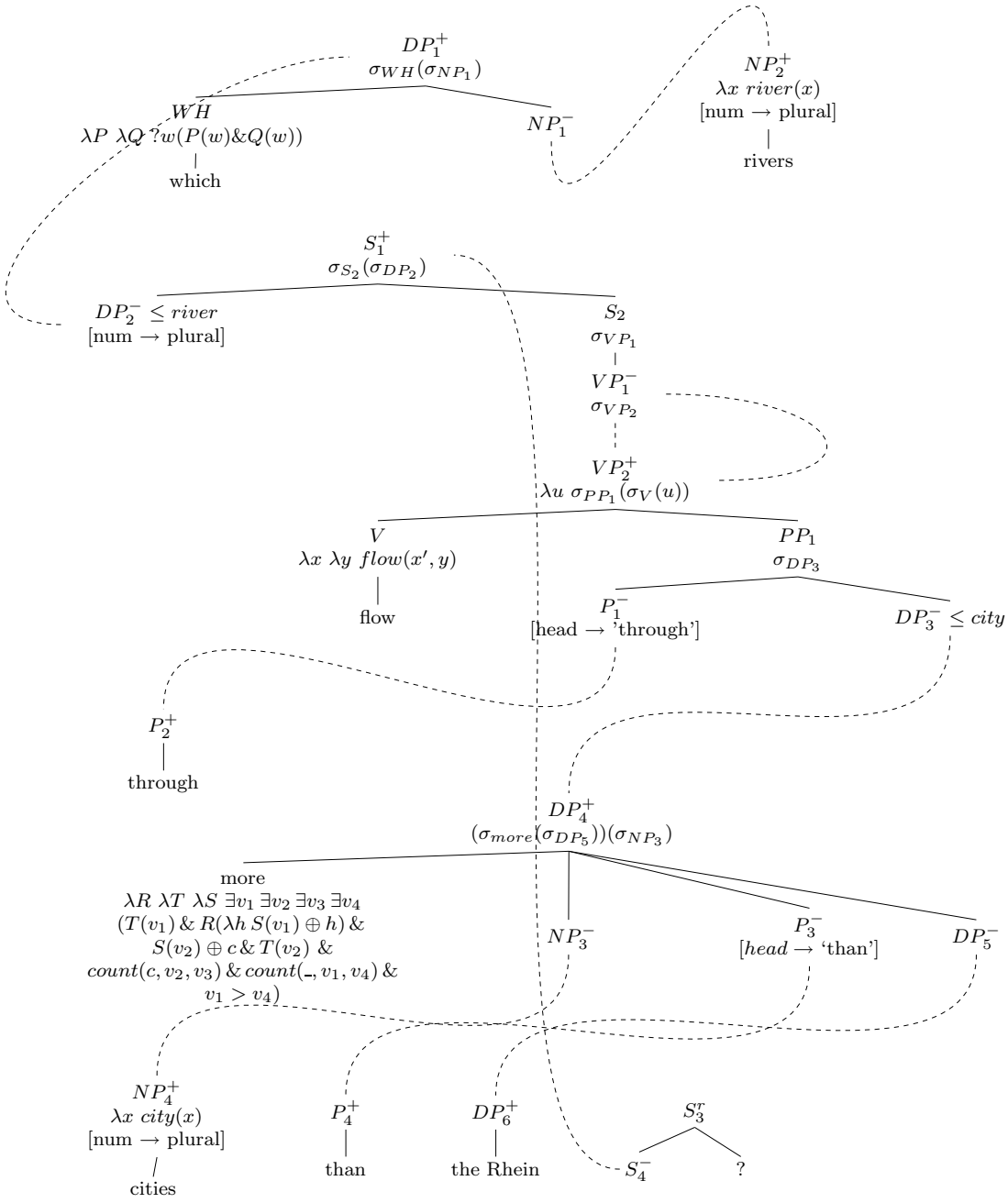


Figure 3.2: Elementary trees for *"Which rivers flow through more cities than the Rhein?"*

Chapter 4

Domain Adaption

In our system, we pursue an approach in which the domain lexicon is constructed in interaction with the user, whose task is to map relations in the knowledge base to appropriate verb and noun subcategorization frames, adjectives, etc. Before explaining in detail the underlying model which allows a user to create a domain-specific lexicon and thus customize the system to a certain knowledge base, it is important to mention that the overall lexicon of the system has a bipartite structure consisting of:

- a *domain-independent lexicon*, containing the semantic representations for determiners (*a, the, every, most, ...*), wh-pronouns (*who, what, which, where*) as well as certain spatio-temporal prepositions (*on, in, at, before, ...*),
- a *domain-specific lexicon*, defining the meaning of verbs, (relational) nouns and adjectives occurring in the domain, and containing lexical entries and the semantics of instances and concepts, which are typically represented linguistically as proper nouns and nouns, respectively.

The *domain-independent lexicon* is, as the name suggests, independent of any domain as it specifies the meaning of words occurring in several domains and with a constant meaning across these. This is the case for determiners, wh-pronouns and prepositions. The semantic representations of the words in this domain-independent lexicon thus make reference to domain-independent categories as given for example by a foundational ontology such as DOLCE [Masolo et al., 2003]. This assumes obviously that the domain ontology is somehow aligned to the foundational categories provided by the foundational ontology. The obvious benefit of such a modular design of the lexicon is that the meaning of closed-class words such as prepositions, wh-pronouns or determiners are available independently of any domain ontology and need not to be specified for every different domain the system is applied to. A more detailed description of the general benefits and rationale of such a modularized approach can be found in [Cimiano and Reyle, 2006].

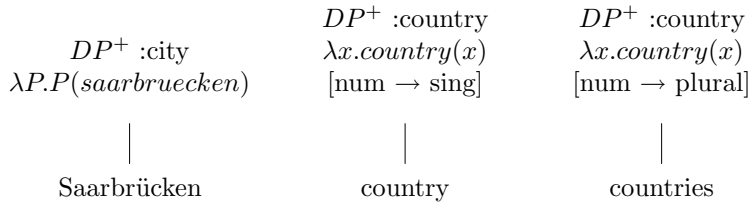


Figure 4.1: Elementary trees automatically generated from the KB

The *domain-specific lexicon* is partially derived fully automatically from the domain ontology loaded into the system without any manual intervention. In fact, the system reads in all the concepts and instances of the ontology and relies on their labels to generate appropriate grammar trees representing these. Obviously this assumes the availability of labels for each concept and instance in the ontology. However, in general it is regarded as good practice to include such labels into the ontology to enable human inspection. For the generation of nominal trees on the basis of concepts, we use a lexicon with morphological information to generate the appropriate plural form. This lexicon was generated on the basis of Tree Tagger's tagger lexicon [Schmid, 1994].

For illustration, Figure 4.1 shows the elementary trees which are automatically generated from the instance *Saarbrücken* as well as the concept *country*.

The other part of the domain-specific lexicon component has to be generated by the user is the *domain-specific lexicon*, in which verbs, adjectives and relational nouns are mapped to corresponding relations specified in the domain ontology. The domain-specific lexicon is actually the most important one as it is the one specifying the mapping of linguistic expressions to domain-specific predicates. It is important to emphasize that our natural language interface does not require any sort of pre-encoded grammar as input of the system. The grammar underlying the ORAKEL system consists exactly of the union of the trees in the domain-independent lexicon, the ontological lexicon and the domain-specific lexicon. Thus, the task of the user is to actually provide a domain-specific grammar to the system. As this is a difficult task - compare the discussion of syntactic variants in Section 3.1 - in our natural language interface we implement an approach in which the user simply instantiates subcategorization frames and maps these to domain-specific relations in the ontology. Actually, the linguistic subcategorization frames as well as the relation types are organized in a type hierarchy, such that only structures of compatible arity are mapped onto each other. As shown in Figure 4.2, in our type system we distinguish between binary, ternary and quaternary subcategorization frames which can be mapped to binary, ternary and quaternary relations, respectively[1].

[1]Note that there is no principled limit to the arity of relation. However, according to our experience considering relations of up to 4 suffices to cover most examples in practice.

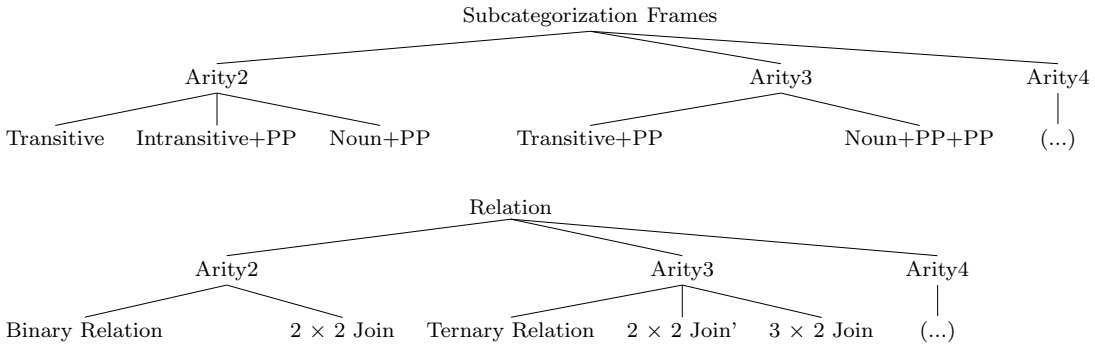


Figure 4.2: Type hierarchies of linguistic templates and relations

Examples for binary subcategorization frames are transitive verbs, intransitive verbs with a prepositional complement, relational nouns with one prepositional complement as well as participles with prepositional complements:

- transitive: verb(subject,object), e.g. *border*
- intransitive + prepositional complement: verb(subject, prep:pobject), e.g. *flow through*
- noun + pp: noun(prep: pcomp), e.g. *capital of*
- participle+pp: participle(prep: pcomp), e.g. *located in*

For example, the user could create the following mappings for a geography knowledge base:

location(pcomp(of): x)	→	locatedIn(x: location, y: location) (1)
inhabitants(pcomp(of): x)	→	inhabitants(x: location, y: integer) (2)
live(subj: x,pobj(in): y)	→	inhabitants(y: state/city, x: integer) (3)
capital(pcomp(of): x)	→	capital(x: city, y: state) (4)
length(pcomp(of): x)	→	length(x: river, y: integer) (5)
flow(subj: x, pobj(through): y)	→	flow_through(x: river, y: city) (6)
pass(subj: x, pobj(through): y)	→	flow_through(x: river, y: city) (7)
pass(subj: x, pobj(through): y)	→	located_at_highway(y: city, x: highway) (8)
height(pcomp(of): x)	→	height(x: mountain, y: integer) (9)
border(subj: x, obj: y)	→	borders(x: location, y: location) (10)
located(pcomp(in): x)	→	locatedIn(y: location, x: location) (11)

First of all, note that though these mappings may seem straightforward, they are indeed crucial for ORAKEL to generate a full domain-specific grammar mapping linguistic expressions to appropriate semantic representations. How should ORAKEL in fact know that the relation *border* is best expressed with a transitive verb with the same name? How should it know that the *capital* relation should best be expressed by the noun *'capital (of)'*? Though

simple heuristics based on matches between relation names and verbs or nouns might be applied, they will in general not suffice to cover all the possible lexical variations one can use to ask for a specific relation. Language is to variable to be captured by such easy heuristics. Further, it is crucial to determine the order in which the arguments of the relation map to arguments of the linguistic predicate, e.g. the verb or noun in question. Instead of building on an automatic, heuristic, and therefore error-prone process, in ORAKEL we build on a more controlled approach in which users can specify lexical variants (with some support though) as well as the correct order in which the arguments map onto each other. Examples of mappings which are not as straightforward are (3) and (7). The 3rd mapping is interesting in that it provides a non-straightforward lexical variant for asking for the inhabitants of a city. The 7th mapping introduces a further non-obvious lexical variant to ask for the *flow_through* relation between rivers and cities. By this, we introduce a lexical ambiguity into the lexicon, as *'pass through'* can denote either the *flow_through* relation between rivers and cities as well as the *located_at_highway* relation between highways and cities. Actually, it is not always the case that the domain of a relation is mapped to the subject and the range to the object in the corresponding verb subcategorization frame. Such an example is provided by mapping (8) where the subject and object of *'pass through'* are mapped to the range and domain of *located_at_highway*, respectively. It is therefore necessary that the user also specifies the order in which the relation's arguments map to the ones of the subcategorization frame. For the noun subcategorization frames, the argument of the relation which has not been mapped to the *pcomp* position – the y-argument in the above examples – is stored separately from the actual frame as it will be normally expressed in form of a copula[2] construct such as *"What is the length of the Rhein?"*. Note that this holds also for participles which are also typically used in copula constructs, e.g. *"Where is Karlsruhe located in?"*.

Further, for nouns complemented by the preposition *'of'*, the system also generates trees allowing to ask for the corresponding relation using the verb *'have'* (see the examples below). For methods such as *capital_of*, which do not have a datatype such as a string or an integer as range, and which have been mapped to a noun+pp, the generator does not only generate relational noun phrases such that one can ask: *"What is the capital of Baden Württemberg?"* using a copula construct, but also a form in which the argument mapped to the *pcomp* position is existentially quantified over. This allows to ask a question like *"Which rivers flow through a capital?"* For verbs, it generates the active, passive and verb-last forms, but also relative clauses complementing a noun phrase (see the tree family for a transitive verb described in Section 3.1). On the basis of the above example mappings, the system then generates elementary trees, such that it is able to interpret the following questions:

[2] A copula is an expression involving the verb *'be'* and linking the subject to some property or object.

What is the location of Stuttgart? (1)
How many inhabitants does Baden Württemberg have? (2)
How many people live in Karlsruhe? (3)
What is the length of the Rhein? (5)
What is the capital of Baden Württemberg? (4)
Which river flows through the capital of Baden Württemberg? (4+6)
Which rivers flow through a capital? (4+6)
What is the length of the Rhein? (5)
Which river flows through the most cities? (6)
Which river flows through a state which borders Baden Württemberg? (6+10)
Which river passes through München? (7)
Which highways pass through Berlin? (8)
What is the height of the Zugspitze? (9)
Which countries does Baden Württemberg border? (10)
Which countries are bordered by Baden Württemberg? (10)
Which countries border Baden Württemberg? (10)
Which state borders the most countries? (10)
Where is Karlsruhe located in? (11)

Binary relations with an integer as range are special types of relations which can also be mapped to adjectives by specifying (i) the base, (ii) the comparative, and (iii) the superlative form of the adjective, additionally indicating whether it denotes a positive or negative scale (this is similar to the approach in TEAM [Grosz et al., 1987]). For example, the adjectives *'big', 'long' and 'high'* are mapped to the the relations *inhabitants*, *length* and *height*, respectively:

adj(big,bigger,biggest,positive)	→	inhabitants(city,integer) (Adj1)
adj(long, longer, longest,positive)	→	length(river,integer) (Adj2)
adj(high,higher,highest,positive)	→	height(mountain,integer) (Adj3)

This then allows to ask the following questions:

How long is the Rhein? (Adj2)
How high is the Zugsitze? (Adj3)
How big is Karlsruhe? (Adj1)
Which is the longest river? (Adj2)
Which river is longer than the Rhein? (Adj2)
Which is the highest mountain? (Adj3)
Which cities are bigger than Karlsruhe? (Adj1)

The positive/negative distinction is thus necessary to generate the correct semantics for comparative and superlative adjectives. In fact, *'big', 'long'* and *'high'* are positive adjectives in our sense, while *'small'* is an example of a negative adjective. In general, specifying the semantics of adjectives in base form is a quite delicate issue as an adjective such as *'big'* actually denotes a fuzzy set in the sense of Zadeh [Zadeh, 1975]. However, we need to specify the semantics of adjectives in order to answer queries such as *"Which rivers flow through big cities?"*. The solution adopted in ORAKEL is to expect a

definition of the semantics of an adjective in terms of a rule, e.g.

$$\forall x\ big(x) \leftarrow city(x) \wedge inhabitants(x, y) \wedge y > 500.000$$

It is important to emphasize that currently ORAKEL can only handle scalar adjectives such as *'big'*, *'high'*, *'long'*, etc. In particular, it can not deal with non-scalar adjectives such as *'German'*, which would need to be translated into a corresponding relation in which a specific value is inserted. The adjective *'German'*, for example, could be translated into the expression $\lambda x\ locatedIn(x, Germany)$.

In order to allow a user for specifying the above described mappings, we have created a tool called FrameMapper which supports the user graphically in performing the mappings. Currently, there is no graphical support within the FrameMapper system to describe the semantics of adjectives in this way. Thus, the specification of the semantics of adjectives needs to be specified by hand, which is currently a bottleneck in our system. However, we are currently devising methods to also support the user in specifying the semantics of adjectives with the FrameMapper tool in an intuitive way without having to define a logical rule. Note that the specification of the semantics of the comparative and superlative version of an adjective does not face this problem as only relative comparisons between elements along the underlying scale need to be conducted. Thus, specifying the attribute the adjective maps to as well as whether the scale is positive or negative is sufficient.

As shown in the type hierarchy depicted in Figure 4.2, the mapping model is not restricted only to binary relations. Subcategorization frames can also be mapped to joins of several relations, e.g. a subcategorization frame of arity 2 can also be mapped to two binary relations joined at a given position (2×2-Join in Figure 4.2), a subcategorization frame of arity 3 can be mapped either to a simple ternary relation, a join of two binary relations in which the joined position is also mapped to an argument in the frame (2×2-Join' in the Figure) or to a join of 3 binary methods (3×2-Join in the Figure), etc. Hereby *Join'* denotes a join in which the joined position has also been mapped to an argument in the subcategorization frame while for *Join* this is not the case. This explains why $n \times 2$ *Join'* joins have an arity of $n + 1$ while $n \times 2$ *Join* joins have an arity of n.

The reason for introducing such an elaborated type system is the fact that linguistic expressions in many cases do not correspond directly to one relation in the knowledge base, but express a combination of different relations in the knowledge base, which can be expressed through joins.

As a more complex example, assume the following relations given in the knowledge base: *author(paper,author)*, *title(paper,title)*, *year(paper,string)*. If we create a 3×2 Join by joining the paper position of the three relations, we can map this ternary relation to a transitive verb *'publish'* with a prepositional complement introduced by the preposition *'in'* such that we can ask a question like *"Who published which paper in 2002?"* (see also the discussion of this join in Chapter 5).

Summarizing, the crucial aspect here is actually the fact that the domain-specific grammar necessary for understanding domain-specific expressions is generated in the background as a byproduct of a user interacting with the system and mapping subcategorization frames onto appropriate relations in the knowledge base. Thus, no pre-encoded grammar is actually needed in the system. In order to map relations defined in the ontology to appropriate subcategorization frames, users are supposed to use the *FrameMapper* lexicon creation frontend, which allows to select a relation and to create corresponding subcategorization frames. The ontological restrictions on the concepts which can be used at the different argument positions of the relation will then be used as selectional restrictions in the subcategorization frames and exploited for disambiguation (compare Section 3.6). After the user has assigned all the relations to corresponding subcategorization frames or adjectives, he can export the lexicon, which can then be used by the natural language interface to answer users' questions against the knowledge base. In our model, we do not expect a lexicon engineer to model the lexicon in one turn from scratch, but assume that the lexicon is created in several iterations. After the domain expert has created a first version of the lexicon, the system is deployed. The domain expert gets presented the questions which the system failed to answer and the process is iterated. Our hypothesis in fact is that with such an iterative method, the quality of the lexicon can be constantly improved. We will present experimental evidence for this hypothesis in Chapter 6. Before presenting the results of our experiments in Section 6, in the following chapter we describe FrameMapper's graphical user interface.

Chapter 5

Graphical User Interface

Figure 5.1 shows a screenshot of FrameMapper's graphical user interface. It shows how a user is mapping the *flow_through* relation to the intransitive verb *'flow'* featuring a prepositional complement introduced by the preposition *'through'*. The figure shows the three main panes of *FrameMapper*. In the top pane, the user sees the relations specified in the ontology. In the second pane, s/he can see the different subcategorization frames assigned to the active relation. In the third pane, he sees a graph visualization of the current subcategorization frame and of the selected relations. He can then graphically map the arguments of the frame to the ones of the selected relation(s). In the GUI screenshot in Figure 5.1, the user has already mapped the intransitive verb *'pass'* with a prepositional complement introduced by *'through'* to the *flow_through* relation (this can be seen in the middle pane). Currently, he is also mapping the intransitive verb *'flow'* with a prepositional complement introduced by the preposition *'through'* to the same relation. In particular, he has already mapped the subject position of the verb *'flow'* to the domain of the *flow_through* relation and the prepositional complement to the range position of the same relation. Further, in the screenshot he has already entered the appropriate preposition *'through'* in the graph representing the subcategorization frame and is currently editing the verb, specifying that its base form is actually *'flow'*. With this information, the system can in the background generate all the grammatical variations of the intransitive verb *'flow'*, thus allowing to ask for the *flow_though* relation in a variety of ways. In order to add a further verb, the user simply has to instantiate a new subcategorization frame and perform the mapping again. The newly created subcategorization frame would then be added to the list of those subcategorization frames already created for the active relation(s) in the middle pane. In order to ease the process of adding lexical variants, we have also integrated the WordNet lexical database [Fellbaum, 1998] with the purpose of automatically suggesting synonyms for the verb or noun currently edited. For this purpose, we only consider the first sense of each word, suggesting each of the words contained in the corresponding synset to the user as lexical variants on demand in the form of a check-box. Each selected synonym is then used to gen-

erate subcategorization frames only differing in the lexical element. However, this functionality was added recently and not used in the experiments described in Chapter 6.

It is important to mention that the type hierarchy described in the previous section is used to constrain the subcategorization frames offered to the user. For example, if he selects one binary relation, he will only be able to instantiate a transitive, intransitive+PP or noun+PP subcategorization frames. Adjectives can only be created for relations with an integer as range.

Note that the user can also select various relations and carry out joins between them to specify more complex mappings involving more than one relation. Figure 5.2 shows a screenshot of the GUI in which the user has chosen the three relations *author(publication,person)*, *title(publication,string)* and *year(publication,string)*, all joined through their domains, i.e. through the publication. The user has further instantiated a subcategorization frame for the transitive verb *'publish'* featuring a direct object as well as a prepositional complement introduced by the preposition *'in'*. Further, he has mapped the range of the author*(publication,person)* relation to the subject position, the range of the *title(publication,string)* relation to the object position as well as the range of the *year(publication,string)* to the prepositional complement. This mapping would then allow to ask a question like *"Who published which paper in 2002?"*.

In general, the user can export the lexicon, which can then be loaded into the ORAKEL natural language interface, but s/he can also import an already created mapping lexicon to add more subcategorization frames, thus supporting our iterative lexicon generation model.

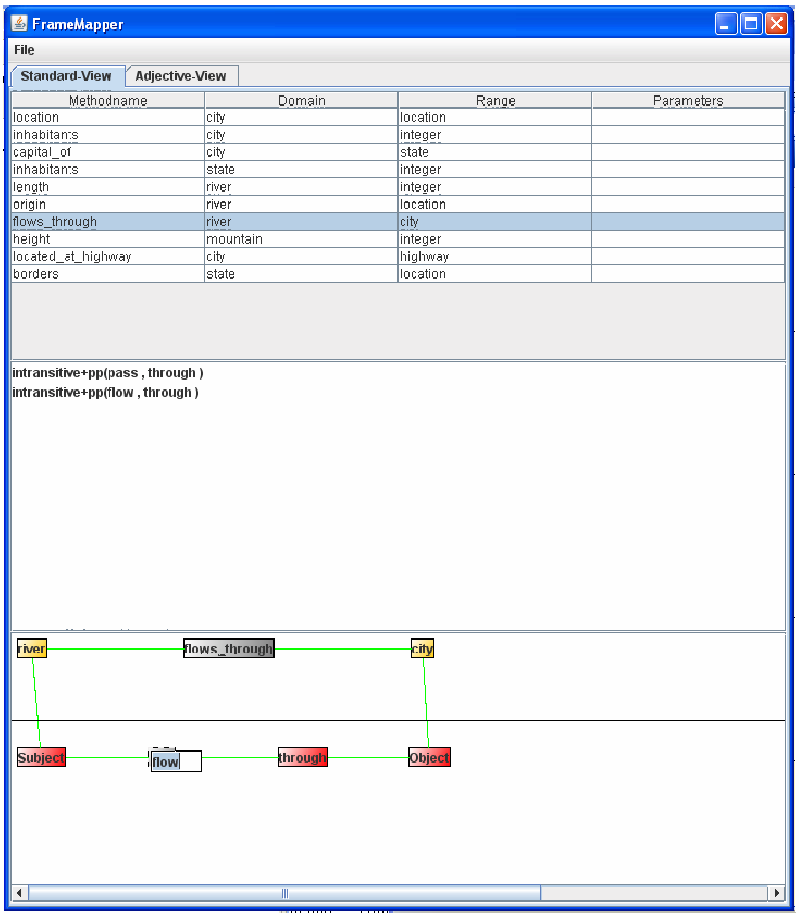


Figure 5.1: GUI of FrameMapper showing a simple mapping for the geographical domain.

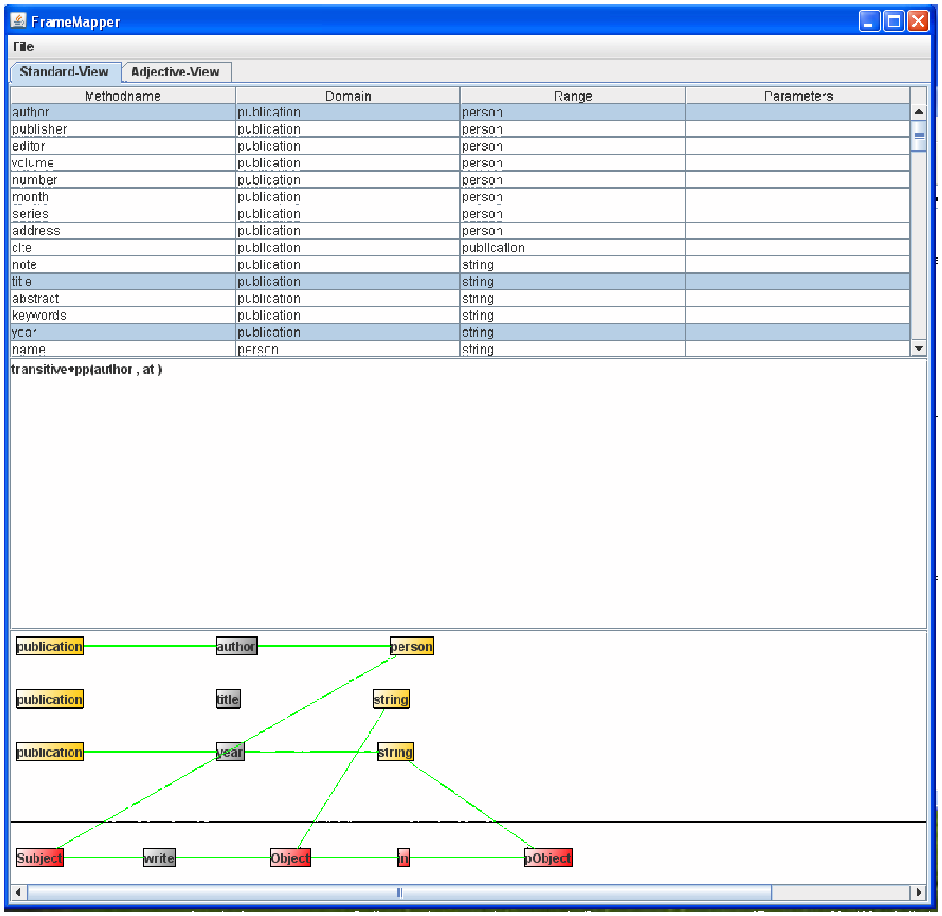


Figure 5.2: GUI of FrameMapper showing a more complex mapping involving joins for the academic domain.

Chapter 6

Experiments

In this chapter, we first present the settings and results of our experiments, which have been carried out on two different domains showing that the ORAKEL system can be adapted to different domains without major efforts. First, we present a user study carried out with a knowledge base and corresponding ontology containing facts about German geography. The aim of this study was to demonstrate that computer scientists without any NLP expertise can indeed generate domain-specific lexica for the ORAKEL system without major difficulties. Second, we provide some statistics demonstrating that the system has potentially a reasonable linguistic coverage. The results of the first study have been partially presented also in [Cimiano et al., 2007] but presented here in more detail. In this paper, we additionally discuss a case study carried out at British Telecom in which the ORAKEL natural language interface was successfully applied to offer enhanced search over a digital library.

6.1 User study

The aim of the user study was to show that computer scientists without any NLP expertise can indeed generate reasonable domain-specific lexicons for the ORAKEL natural language interface. The study also provides first evidence that our iterative approach is indeed feasible.

The knowledge base used for the experiments contains geographical facts about Germany. In particular, it contains states, cities, rivers and highways in Germany, as well as the name of the neighboring countries. It is a small knowledge base handcrafted by students at our department independently of the experiments described here. The knowledge base contains the number of inhabitants of each state and city as well as the capital of each state. For rivers and highways, it contains information about the cities they pass. For rivers, it additionally contains their origin as well as length. It also contains mountains and their heights. Overall, the knowledge base comprises 260 entities: 108 highways, 106 cities, 18 rivers, 16 states, 9 (bordering) countries

and 2 (bordering) seas as well as one mountain peak, i.e. the *Zugspitze*. The relations defined in the ontology are the following ones (given in F-Logic style notation):

```
city[locatedIn => location].
city[inhabitants => integer].
state[inhabitants => integer].
state[borders =>> location].
city[located_at_highway =>> highway].
river[length => integer].
river[origin => location].
river[flows_through => city].
mountain[height => integer].
city[capital_of => state].
```

Here, => denotes that the relation is functional, i.e. it can have at most one instance as range, and =>> denotes that there can be more than one instance as range of the relation.

The user study involved one of the authors of this paper, as well as 26 additional test persons from 4 different institutions, both academic and industrial. Of these 26 test persons, 25 were computer scientists and 1 a graphic designer, most of them without any background in computational linguistics. The role of the author as well as two of the other participants was to construct a lexicon each, while the rest played the role of end users of the system. We will refer to the author as A and the other two participants constructing a lexicon as B and C. While A was very familiar with the lexicon acquisition tool, B and C were not and received 10 minutes of training on the tool as well as 10 minutes explanation about the different subcategorization types, illustrated with general examples. Whereas A constructed a lexicon in one turn, B and C constructed their lexicon in two rounds of each 30 minutes. In the first round, they were asked to model their lexicon from scratch, while in the second round they were presented those questions which the system had failed to answer after the first round consisting of 4 sessions with different users. They were asked to complete the lexicon on the basis of the failed questions. Overall, they thus had one hour to construct the lexica. The 24 persons playing the role of the end users also received instructions for the experiment. They received a document describing the experiment, requiring them to ask at least 10 questions to the system. Further, the scope of the knowledge base was explained to them. They were explicitly told that they could ask any question, also involving negation and quantification, with the only restriction that it should begin with a *wh*-pronoun such as *which, what, who, where* as well as *how many* or *how + adjective*. For each answer of the system, they were asked to specify if the answer was correct or not. The results are thus reported in the following as *recall*, i.e. the number of questions answered correctly by the system divided by the total number of questions asked to the system. Excluded from this were only questions with spelling errors or which were obviously ungrammatical, as well as questions which were clearly out of the scope of the knowledge base. We

Lexicon	Users	Rec. (avg.)	Prec. (avg.)
A	8	53.67%	84.23%
B (1st lexicon)	4	44.39%	74.53%
B (2nd lexicon)	4	45.15%	80.95%
C (1st lexicon)	4	35.41%	82.25%
C (2nd lexicon)	4	47.66%	80.60%

Table 6.1: Results for the different lexica

also give the *precision* of our system as the number of questions for which the system returned a correct answer divided by the number of questions for which it returned an answer at all. Note that precision and recall are defined here in line with [Popescu et al., 2003] and not in the standard information retrieval sense (cf. [Baeza-Yates and Ribeiro-Neto, 1999]).

Table 6.1 shows these results for each of the lexicon constructors and the two iterations.

The first interesting conclusion is that, for both B and C, there is an increase in recall after the first round. Thus, the results show that our iterative methodology to lexicon customization is indeed promising. The involved users also confirmed that it was easier to extend the lexicon given the failed questions than creating it from scratch. The second interesting result is that the lexicons created by B and C show a comparable recall to the lexicon developed by A. In fact, we found no significant difference (according to a Student's t-test at an α-level of 0.05) between the results of B's lexicon ($p = 0.32$) and C's lexicon ($p = 0.15$) compared to A's lexicon. This shows that our lexicon acquisition model is in fact successful. In general, the results have increased after the second iteration, with the exception of a slight drop in precision for user C at the second round. We expect that further iterations will continuously improve the lexica. This is, however, subject to further analysis in future work.

6.2 Question Analysis

Having shown that domain experts are able to map relations in a knowledge base to subcategorization frames used to express them, an important question is to determine how big the coverage of the different subcategorization frames is with respect to the questions asked by the end users. Overall, the end users asked 454 questions in our experiments (actually much more than the number of questions requested). Table 6.2 summarizes the constructions used together with illustrating examples, giving their percentage with respect to the total number of questions. The results show that in principle, assuming that the lexicon is complete, with our basic subcategorization frames *transitive*, *intransitive+pp*, *np* and *adj* as well the constructions automatically generated from these (marked with a '*' in the table), we get a linguistic coverage of over 93%[1]. This shows

[1]The constructions marked with a − were added after the user study described before.

construction	#	%	Example
intransitive+pp (*)	169	37.22%	How many cities *does* the A1 *pass through* ?
transitive (*)	56	12.33%	How many states *does* Baden Württemberg *border* ?
be+np (*)	102	22.47%	What *is the capital of* Bayern ?
be+adj (*)	22	4.85%	How *long is* the Donau ?
be+pp (+)	22	4.65%	Which cities *are in* Bayern ?
be+dp (*)	18	3.96%	Where *is* Düsseldorf ?
be+np (poss) (-)	1	0.22%	What *is* Bayern*'s capital* ?
be+participle (-)	12	2.64%	Which cities *are located in* Bayern ?
be+np (inv) (-)	4	0.88%	Which state *is* München *capital of* ?
be+np (superlative) (*)	2	0.44%	What *is the capital of* Bayern ?
be+comp (*)	7	1.54%	Which cities *are bigger than* Frankfurt ?
passive (*)	1	0.22%	Which states *are bordered by* at least 2 countries ?
have (*)	32	7.04%	Which states *have* more inhabitants than Hessen ?
Number of queries:	454	100%	

Table 6.2: Usage of constructions in main clause (in percent with respect to the total number of questions)

that it is indeed feasible to focus on a few subcategorization types. The intelligence lies anyway in the generation of the corresponding elementary trees from the subcategorization frames. The generation, however, remains totally opaque to the user. For the constructs marked with '+', we additionally need to specify the semantics of prepositions with respect to domain-independent relations. For example, *'in'* maps to the relation *locatedIn* in our system. This is assumed to be a relation which is available in any domain. For other domains however, *'in'* might also have a temporal interpretation which can be formalized with respect to the DOLCE foundational ontology ([Cimiano and Reyle, 2006]). Constructions which were added after the experimental evaluation are the ones marked with a '-', i.e. the *be+np (possessive), be+np (inverse)* and *be+participle* constructions, which account for around 3.75% of the cases. Considering the first two is merely a question of generation of the appropriate elementary trees and was accomplished in a straightforward way. For the third construct, the subcategorization types had to be extended to participles subcategorizing a prepositional phrase, e.g. such as *located in*.

6.2.1 Runtime evaluation

In addition to the above experiments, we also performed a runtime analysis of the system[2]. Figure 6.1 shows the average number of seconds needed by ORAKEL to process the question, the number of seconds taken by the OntoBroker inference engine (see [Decker et al., 1999]) to answer the logical query as well as the sum of both. These values are shown grouped and averaged over the sentences up to a certain length (in the number of words). Four observations are straightforward. First, we can observe that all questions are processed and answered within 0.6 seconds at most. Second, we observe that the time taken to process an input question increases proportionally to the number of words in the sentence. Third, it is also interesting to see that the time OntoBroker needs to

[2]This analysis was performed on a personal computer with a 2GHz processor.

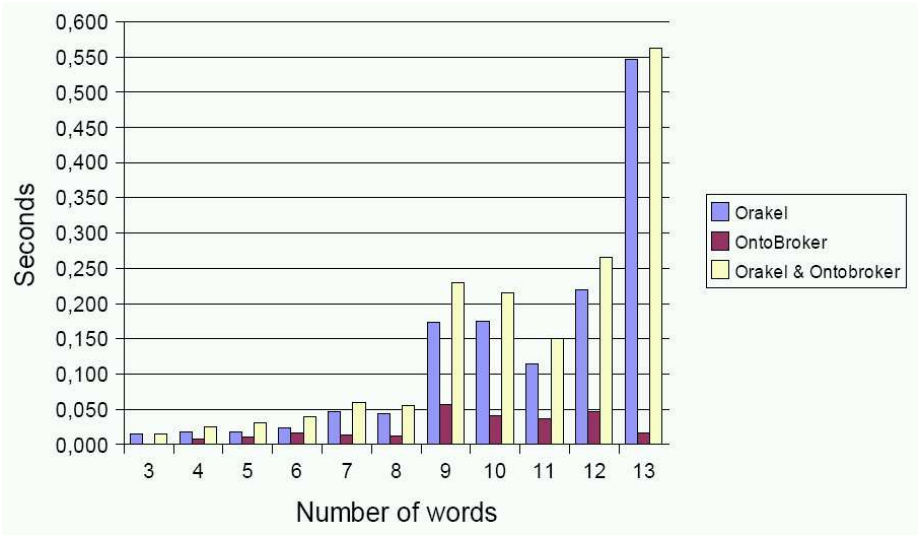


Figure 6.1: Average runtime for ORAKEL and OntoBroker over number of words

answer the queries also increases with the number of words in the input sentence (with the exception of the last value), which shows that the longer questions are not only hard to process but also to answer from an inferencing point of view. Finally, it is also interesting to observe that the time taken by OntoBroker to answer a query can be more or less neglected with respect to the time taken by ORAKEL to translate the question. With respect to all questions, the average time taken by ORAKEL to process a question is 0.13s, for OntoBroker it is 0.02s and for both together it is 0.15s. This shows that the performance of the system is indeed good enough to answer questions in real time.

6.2.2 Real-world application

As a further experiment, our approach has been applied within the British Telecom (BT) case study in the context of the SEKT project[3]. In this case study the aim was to enhance the access to BT's digital library by a natural language interface. BT's digital library contains metadata and fulltext documents of scientific publications. The knowledge base considered within this case study is several orders of magnitude larger than the one considered in the context of the experiments carried out on the geographical domain. The ontology used to describe the metadata is the PROTON ontology[4], which consists of 252 classes and 54 relations. While the PROTON ontology (the schema of the data) is stored in an OWL ontology in this scenario, all the publication metadata are

[3]http://www.sekt-project.com/
[4]http://proton.semanticweb.org/

stored in a database. The schema of the database, however, has been mapped to the PROTON ontology, such that queries to the ontology are evaluated by taking into account the metadata in the database. The knowledge base contains metadata about 67015 authors, 17174 topics and 33501 documents (journal articles, conference papers, conference proceedings, periodicals and books). Further, there are 66870 instances of the *AuthorOf* relation and 165089 instances of the *isAboutTopic* relation. As the data size is indeed several orders of magnitude larger compared to our geography domain, in the context of this use case it was totally unfeasible to generate a grammar entry for each value in the database. Therefore, we performed a slight change to the ORAKEL system to allow to dynamically create grammar trees at query time for names of instances. This was achieved by considering every sequence of upper-case words as a potential candidate for an instance, generating appropriate elementary trees at runtime. For example, for an unknown sequence of words such as "John Davies", the following elementary tree would be generated at runtime:

$$\begin{array}{c} \mathrm{DP}^{+} \\ \lambda P \exists x\ P(x) \wedge label(x, \text{"}JohnDavies\text{"}) \\ | \\ \text{John Davies} \end{array}$$

The ontological lexicon is thus generated only for concepts in the ontology in this scenario, while the part of the lexicon containing the instances is generated dynamically. This move was important to ensure efficiency in this setting.

A graduate student and a PhD student spent overall approx. 6 hours creating a lexicon for a subset of PROTON relevant for the digital library using FrameMapper with the result that queries about authors, topics etc. about documents could be answered successfully against the BT ontology and database. Examples of such questions are:

- What conference papers did John Davies write?
- Which conference papers do you know?
- Who wrote which document?
- Who wrote "The future of web services"?
- Who wrote about "Knowledge Management"?
- What article deals with Photography?
- Which journal articles were written by whom?
- What are the topics of "The future of web services"?
- Which conference papers were classified as religion?
- Which articles are about "Intellectual Capital"?

Iteration	Rec. (avg.)	Prec. (avg.)
1	42%	52%
2	49%	71%
3	61%	73%

Table 6.3: Results for the different iterations

- What articles were written by Lent and Swami?
- Who is the author of a document that talks about which concept?
- Who wrote which articles about what?
- Which documents are about F-Logic and Insurance?

Further, ORAKEL was also modified to process quoted text by matching it against a text index of the metadata in the database using a special purpose predicate *match*. The question *"What articles are about "Intellectual Capital"?"* is translated into the following SPARQL query:

```
SELECT  ?w16 WHERE {
   ?w16 rdf:type <http://proton.semanticweb.org/2005/04/protonu#Article> .
   ?w16 <http://proton.semanticweb.org/2005/04/protont#hasSubject> ?x17  .
   ?x17 rdfs:label x18 .
   match(x18,"Intellectual Capital")
}
```

Furthermore, an evaluation of the ORAKEL system over several iterations was carried out with the BT digital library. As in the experiments described above, the end users received written instructions describing the conceptual range of the knowlege base, asking them to ask at least 10 questions to the system. In each of three iterations, 4 end users asked questions to the ORAKEL system and the graduate student updated the lexicon on the basis of the failed questions after the first and second round for about 30 min., respectively. The end users were also asked to indicate whether the answer was correct or not, which allows for the evaluation of the system's performance in terms of precision and recall. The results of the evaluation are presented in Table 6.3, which clearly shows that the second iteration performed much better than the first one, both in terms of precision and recall. In the third iteration, there was a further gain in recall with respect to the second iteration. Overall, this indeed shows that our iterative lexicon model is reasonable and in fact leads to incremental improvement of the system's lexicon.

Overall, the application of ORAKEL to enhance the access to BT's digital library showed on the one hand that, given certain straightforward modifications, our approach can actually scale to much larger knowledge and data bases. Further, the additional use case confirms that the system can indeed be ported between domains in a more or less straightforward way. We refer the

interested reader to [Cimiano et al., 2006] and [Warren and Alsmeyer, 2005] for further details about the case study at British Telecom.

Chapter 7

Related Work

In the 70's and 80's, natural language interfaces were a fashionable research topic (compare [Androutsopoulos et al., 1995] and [Copestake and Jones, 1989]). First companies selling natural language interfaces as products were launched, e.g. *Intellect* from Trinzic, *Parlance* from BBN, *Languageaccess* from IBM, *Natural Language* from Natural Language Inc., *English Wizard* from the Linguistic Technology Corporation[1]. However, it seems that these early systems were not robust or mature enough for application and researchers drifted away from the topic at some stage. In fact, in the 90's there was a significant decrease in the number of publications on the topic. However, as the amount of information, services etc. keeps steadily growing and the electronic devices to access these get smaller and smaller, the necessity for natural language interfaces gains in importance again. Since the new millennium, a new generation of researchers have in fact started again to tackle the task of building natural language interfaces (compare [Popescu et al., 2003], [Lopez and Motta, 2004], [Bernstein et al., 2005], [Minock, 2005] and [Frank et al., 2007]). So we should ask: what has changed since the research in the 70's and 80's? What makes this new generation of researchers confident that the task of building natural language interfaces is feasible? The answer is far too complex to be answered within this overview of related work, but for sure the current wide availability of various resources plays a key role. On the one hand, a plethora of lexical resources (WordNet [Fellbaum, 1998], FrameNet [Baker et al., 1998]), general or upper-level ontologies (Cyc[2], SUMO [Pease et al., 2002], DOLCE [Masolo et al., 2003]), grammars and parsers are available to be used off-the-shelf nowadays, which eases the work considerably. On the other hand, decades of research in databases and knowledge representation have lead to the establishment of *de facto* standards such as the relational model, SQL as query language in the field of databases, OWL [McGuiness and van Harmelen, 2004] and

[1] More recently, other commercial NLIs have appeared, such as *English Query* from Microsoft and *ELF Access* from ELF Software. See `http://www.elfsoft.com/ns/FaceOff.htm` for a qualitative comparison of ELF Access, English Query and English Wizard.

[2] http://www.opencyc.org/

RDF [Brickley and Guha, 2004] as knowledge representation languages as well as SPARQL [Prud'hommeaux and Seaborne, 2006] and other query languages in the fields of Knowledge Representation and the Semantic Web. Thus, researchers nowadays have off-the-shelf data management frameworks as well as well-defined query languages and interfaces at hand to work with. In the 70s, for example, researchers still had to put considerable effort into accessing data distributed in diverse text files (compare [Thompson and Thompson, 1985]). In addition, research in other computer science areas such as graph algorithms has also lead to new ways of tackling the task (compare [Popescu et al., 2003]). Finally, advances in computer infrastructure and platform-independent languages such as Java have lead to the development of high-performance computers as well as eased the portability of programs across platforms (compare the techniques described in [Thompson and Thompson, 1985] to port the system to different languages and computer architectures). It seems that the state-of-the-art in other areas is thus advanced enough to allow tackling the task of building natural language interfaces again from other perspectives. Researchers nowadays are indeed mainly working on robustness, building on – from the natural language understanding point of view – shallow techniques (compare [Popescu et al., 2003] and [Lopez and Motta, 2004]) as well as building hybrid approaches combining deep and shallow analysis (compare [Frank et al., 2007]).

Discussing all the different approaches is out of the scope of this paper. For a detailed review of natural language interfaces, the interested reader is referred to the overviews in [Copestake and Jones, 1989] and [Androutsopoulos et al., 1995], as well as to the evaluation survey in [Ogden and Bernick, 1996]. The latter shows that there have been already many laboratory and field studies of natural language interfaces. However, most of the results reported are neither conclusive nor address the issue of how easy it is for non-NLP experts to adapt the systems to a given domain.

Due to space limitations, we will only discuss three well-known older systems focusing on transportability, i.e. TEAM [Grosz et al., 1987], PARLANCE [Bates, 1989] and ASK [Thompson and Thompson, 1985], as well as six more recent systems: PRECISE [Popescu et al., 2003], Thompson et al.'s system [Thompson et al., 1997], Quetal [Frank et al., 2007], Aqualog [Lopez and Motta, 2004], STEP [Minock, 2005] as well as the system described in [Bernstein et al., 2005].

Early natural language interfaces such as LADDER [Hendrix et al., 1978] were based on so called semantic grammars, which interleaved syntactic and semantic information into one grammar used for parsing and query construction. The non-terminals of the grammar where actually semantic rather than syntactic categories. The underlying grammar was therefore completely tailored to a specific domain and could not be used in a straightforward way for other domains. To overcome the difficulty in porting systems based on semantic grammars such as LADDER, systems such as TEAM, PARLANCE and ASK aimed at supporting the portability of NLIs across domains by database experts.

TEAM's [Grosz et al., 1987] approach to customization consisted in asking questions to a user to acquire linguistic knowledge about certain words, i.e.

verbs, nouns, adjectives etc. as well as their relation to database fields. For a verb, TEAM would for example ask a user for its arguments and whether they are optional or mandatory, for prepositional phrases, for particles and whether they are separable from the verb as well as for different possible types of realization, e.g. passive, unaccusative or dative constructions, etc. TEAM also acquires information about adjectives in a similar way as ORAKEL by asking for the predicate in the knowledge base expressing the corresponding attribute as well for the direction of the scale measured (e.g. positive for *'big'*, negative for *'small'*). Further, TEAM also handles closed-class words by assigning them a domain-independent meaning. In general, TEAM and ORAKEL share three very important aspects:

- the assumption that the knowledge base is independent of the system and should not be changed for the purposes of the natural language interface,
- the requirement that users with knowledge about the domain or underlying database but without any knowledge about formal linguistics or natural language processing should be able to adapt of the system, as well as
- a system design which cleanly separates domain-independent from domain-specific components thus supporting the adaption in a principled way.

Concerning the last point, in TEAM questions are parsed and translated into a general logical form including quantifiers (but also intentional operators and high-order operators, query operators etc.), which is then translated in a second step to the structure of the underlying database. Further commonalities are the fact that both systems support joins of relations in the database or knowledge base and do the crucial distinction between *object* and *datatype* properties, which are called *symbolic* and *arithmetic* fields in TEAM, respectively. Further, TEAM offers some support for type coercion and disambiguating quantifier scope. Concerning the interpretation of light verbs such as *'have'* or nominal compounds, TEAM applies a different strategy than in our approach. While we rely on a generation approach in which *'have'* has as many elementary trees as possible domain-specific interpretations, TEAM tries to map *'have'* to an appropriate database relation while analyzing the sentence. In our approach, the possible meanings of *'have'* are actually determined by the noun+of structures instantiated by the lexicon engineer, for example *capital (of)*, *inhabitants (of)*, etc. This allows then to correctly interpret questions such as *"Which capital does Baden Württemberg have?"* or *"How many inhabitants does Karlsruhe have?"*. Analogous is the treatment of the vague preposition *'with'* in our approach, allowing to ask questions like *"Which is the city with the most inhabitants?"*.

ASK [Thompson and Thompson, 1985] and PARLANCE [Bates, 1989] allowed users to teach new words and concepts even during execution time. However, to our knowledge, the only system of the above which has been evaluated in terms of the time taken to be customized is PARLANCE. According to Bates [Bates, 1989], porting PARLANCE takes between 6-8 person weeks for

databases with between 32 and 75 fields. Such an effort is enormous compared to the one presented in this paper.

Customization to a domain in the system of Thompson et al. [Thompson et al., 1997] is achieved by training a parser using ILP techniques on the domain in question. In particular, a standard shift-reduce parser is assumed and ILP is used to learn parsing control strategies. Such an approach obviously needs training data and Thompson et al. do not discuss if such an approach relying on training data is actually feasible from a usage point of view. The system has been evaluated on two domains (jobs and geography), and achieves very decent accuracy levels between 25 - 70% for the geography domain and between 80 - 90% accuracy for the job domain, depending on the amount of training data used.

The PRECISE system [Popescu et al., 2003] focuses on the reliability of NLIs and is formally proved to be 100% precise, given an appropriate domain-specific lexicon. PRECISE implements an approach based on graph matching in which, essentially, the words in the input question are mapped to database relations, columns and values. For this purpose, so called syntactic markers are removed from the input sentence with the result that only content words remain, which are mapped to tokens representing database relations, columns or values. Hereby a token is a set of words matching a certain database element. PRECISE introduces the notion of *semantically tractable questions*, i.e. questions which have a complete tokenization such that each word is mapped to a distinct token corresponding exactly to one database element and such that every value has been mapped to one attribute and at least one of the value tokens matches a wh-value. The problem is thus reduced to finding a mapping between words and tokens such that attributes are connected to their values. This problem can in fact be formulated in terms of a graph problem and the max-flow algorithm applied to compute a maximal flow with satisfies the above constraints. It is interesting to emphasize that PRECISE also allows ellipsis, that means, attributes to be left out, as in *"Which Chinese restaurants are downtown?"*, where only the relation *restaurants* and two values - *'Chinese'* and *'downtown'* - are specified. The precision of PRECISE of almost 100% on real data is certainly impressive, but in contrast to our approach it only focuses on conjunctive SQL queries. In fact, the questions that it can handle are only a subset of the questions that ORAKEL can handle, which supports arbitrary quantification, Further, our system has also been demonstrated to be very reliable as the precision ranges between 74% and 85%. Strictly speaking, PRECISE does not need any customization, but on the contrary it is not able to handle all the range of questions that ORAKEL can handle. Though they claim to handle also quantification and counting, it is not clear in how far questions like *"Which river flows through every country?"*, *"What is the largest city in the state with the smallest population?"* and *"What river does not traverse the state with the smallest population?"* can be handled. In fact, such questions are strictly speaking not semantically tractable as *'every'* and *'not'* would either be treated as syntactic markers and therefore removed or would not map to any values, columns or relations. For the superlatives *'smallest'* and *'largest'*, similar

remarks apply. The aims of PRECISE and our system are thus complementary, as we have focused on developing an approach by which domain experts can quickly create an appropriate domain-specific lexicon such as needed by any NLI.

The recently presented AquaLog system [Lopez and Motta, 2004] essentially transforms the natural language question into a triple-representation and then relies on similarity measures to map the triples to appropriate relations defined in the ontology. Some basic support for disambiguation is provided in this way. An obvious benefit is that AquaLog does not rely on any sort of customization.

PRECISE and Aqualog are examples of systems relying on lexical matches to map a natural language expression to appropriate knowledge base or database relations. However, such approaches face principled limits. For example, given that the relation between an author and his publications is termed *authorOf*, a question like *"Who wrote which publication?"* could not be analyzed correctly unless we have other knowledge available linking *writing* to an *author*. Though we could imagine that some sort of background knowledge, coming from WordNet for example, is available in the system, there will always be cases in which some piece of knowledge is missing. An approach like ours, where a lexicon engineer specifies the mapping between natural language and the relations defined in the knowledge base, does not face these principled limits as any mapping can be created and the mappings can be directly controlled by the lexicon engineer. On the downside, a lexicon engineer needs to make the effort of creating the mappings.

The STEP system by Michael Minock [Minock, 2005] implements an approach similar to the semantic grammars used within LADDER. Thus, the portability of the system is also restricted. However, the focus of the STEP system is not on portability but on giving a user feedback in form of paraphrases of his question to make sure that the intended meaning of the question has indeed been captured.

The system presented by Bernstein et al. [Bernstein et al., 2005] builds on a controlled language approach as implemented by the ACE (Attempto Controlled English) framework (see [Fuchs et al., 2006]). It requires a transformation from the parser output structures – DRSs [3] (Discourse Representation Structures) – as produced by the Attempto Parsing Engine (APE) to PQL queries formulated with respect to the relations and concepts of an underlying ontology. This transformation needs to be specified by hand by a system engineer. It is thus not clear if the system can indeed be adapted by end users.

In a strict sense, every natural language interface only supports a restricted language which is determined by the underlying grammar as well as the other capabilities of the system. Approaches based on controlled language, however, besides supporting only a restricted language, also give guidelines about how to (syntactically) express meaning in order to avoid ambiguities. This is the approach followed by ACE (Attempto Controlled English). The English sentence *"Every airline owns an aircraft."* is for example semantically ambiguous

[3] See [Kamp and Reyle, 1993].

between two readings in which either the universal quantifier outscopes the existential quantifier or the other way round. In ACE, a user would be requested to write *"Every airline owns an aircraft."* in the first case, while writing *"There is an aircraft that every airline owns."* in the second.

Finally, the approach in the QUETAL system [Frank et al., 2007] implements the mapping from a question to a query via three intermediary stages: i) construction of Robust Minimal Recursion Semantics (RMRS) representation, ii) mapping to domain-independent frame-like structures relying on SUMO, FrameNet and WordNet, as well as iii) construction of so-called *proto-queries*. The use of RMRS[4] (Robust Minimal Recursion Semantics) supports underspecification of scope ambiguities. Further, the approach implements a hybrid technique to interleave shallow and deep processing for the purpose of robustness and to yield a rich semantic analysis of the questions to which several components can contribute, e.g. a HPSG parser, a shallow finite-state parser, a named entity recognizer etc. The proto-queries are abstract database-like queries comparable to our logical forms, but lacking any sort of quantification, negation or counting operators. Counting is for example performed after the answers have been returned by the inference engine. As in our approach, these proto-queries are translated to different query languages, i.e. SQL or SeRQL, the query language implemented in Sesame [Broekstra et al., 2002]. Domain independence is achieved by mapping RMRS structures to domain-independent resources. While in our approach certain closed-class words are mapped to DOLCE, in the Quetal system, RMRSs are mapped to conceptual frames from FrameNet and domain-independent concepts from SUMO. The mapping to the specific knowledge base is achieved by special rules which need to be written for every different knowledge base. These transformation rules thus constitute the adaption mechanism behind the Quetal system. These rules are defined on SUMO and FrameNet structures and thus map domain-independent structures to domain-specific ones. Thus, the domain of these rules is defined in a principled way and remains constant across domains. However, also in the case of the QUETAL system, the question needs to be raised if such transformation rules can be created by an average user. At least, the specification of these rules requires familiarity with SUMO, FrameNet and WordNet. Such a requirement might impose too large of a burden on a naive user and/or domain expert. The system has been successfully implemented on the one hand for a database containing information about Nobel price winners as well as an RDF knowledge base containing information about language technologies, patents, researchers etc. The system has been evaluated on 100 questions from the nobel prize domain. Given an appropriate filtering and voting mechanism of the three best parse trees, a correct proto query is generated in 58% of the cases. Of the cases for which a correct proto query is generated, 74.1% yield a correct answer. The performance of the QUETAL system is thus comparable to the one of our ORAKEL system.

[4]See [Copestake et al., 2006].

Chapter 8

Conclusion and Future Work

We have presented a new model for customizing a natural language interface to a certain domain. In our model, no knowledge about computational linguistics is required to customize the NLI. Our experiments have shown that the very rudimentary knowledge about subcategorization frames needed can be quickly acquired by domain experts. We have conducted extensive experiments with 26 users, mostly computer-scientists with no background in computational linguistics. Two of these were in charge of customizing the NLI, while the remaining played the role of end users. Our extensive experiments with the ORAKEL system clearly corroborate the hypothesis that our model represents a feasible approach for domain adaption. On the one hand, our results have shown that the domain lexica created by the two users in charge of the lexicon creation do not substantially differ from the one created by one of the authors, a computational linguist, with respect to the coverage of the system. In a further real world case study at British Telecom we have further shown that ORAKEL can indeed scale to knowledge bases with thousands of instances. The additional use case has also provided additional proof that our iterative lexicon development methodology is indeed successful. On the other hand, we have demonstrated that, given a suitable mechanism to generate grammatical structures out of the subcategorization frames, by sticking to the few types defined in our system, we achieve more than 90% of linguistic coverage with respect to the questions asked, assuming that the lexicon is complete. Future work will also aim at decoupling our approach from our own parser, thus allowing to use any parser and syntactic theory. This will also allow the possibility of using languages other than English. In addition, we intend to further investigate the possibility of easying adaption across domains relying on the domain-independent meaning specified in foundational ontologies and which can thus be reused across domains. First steps in this direction have been already presented in this paper, but further research is needed to clarify the full potential of this possibility. On a more gen-

eral note, given that the application barrier for natural language technologies is still very high for end users, future work should indeed take up the challenge of developing models by which this barrier can be effectively lowered, thus enabling a wider application and possibly commercialization of natural language processing technologies. The customization model underlying ORAKEL can be certainly regarded as a first step in this direction. Finally, an interesting question for future research is whether the mappings between linguistic expressions and relations defined in the knowledge base can be learned automatically from a corpus. This idea was already preliminary explored in [Cimiano, 2004], but further research is needed to clarify its full potential as well as weaknesses.

Acknowledgements This research has been supported by the following projects: the BMBF project SmartWeb[1], financed by the German Ministry of Education and Research as well as the EU projects Dot.Kom[2], SEKT[3] and X-Media[4]. Thanks to our students Johanna Wenderoth and Laura Goebes for creating the German geography knowledge base. Thanks to all our colleagues from the AIFB, the FZI and ontoprise as well as to Ursula Cimiano, Sofia Pinto and the people from British Telecom for taking part in our experiments.

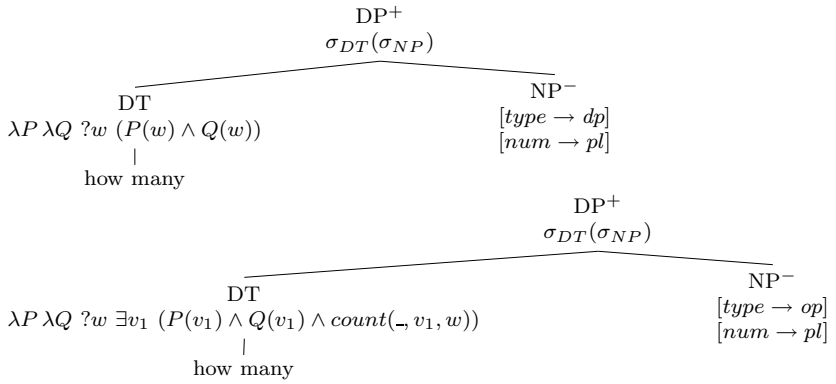
[1] http://www.smartweb-project.de/
[2] http://nlp.shef.ac.uk/dot.kom/
[3] http://www.sekt-project.com/
[4] www.x-media-project.org/

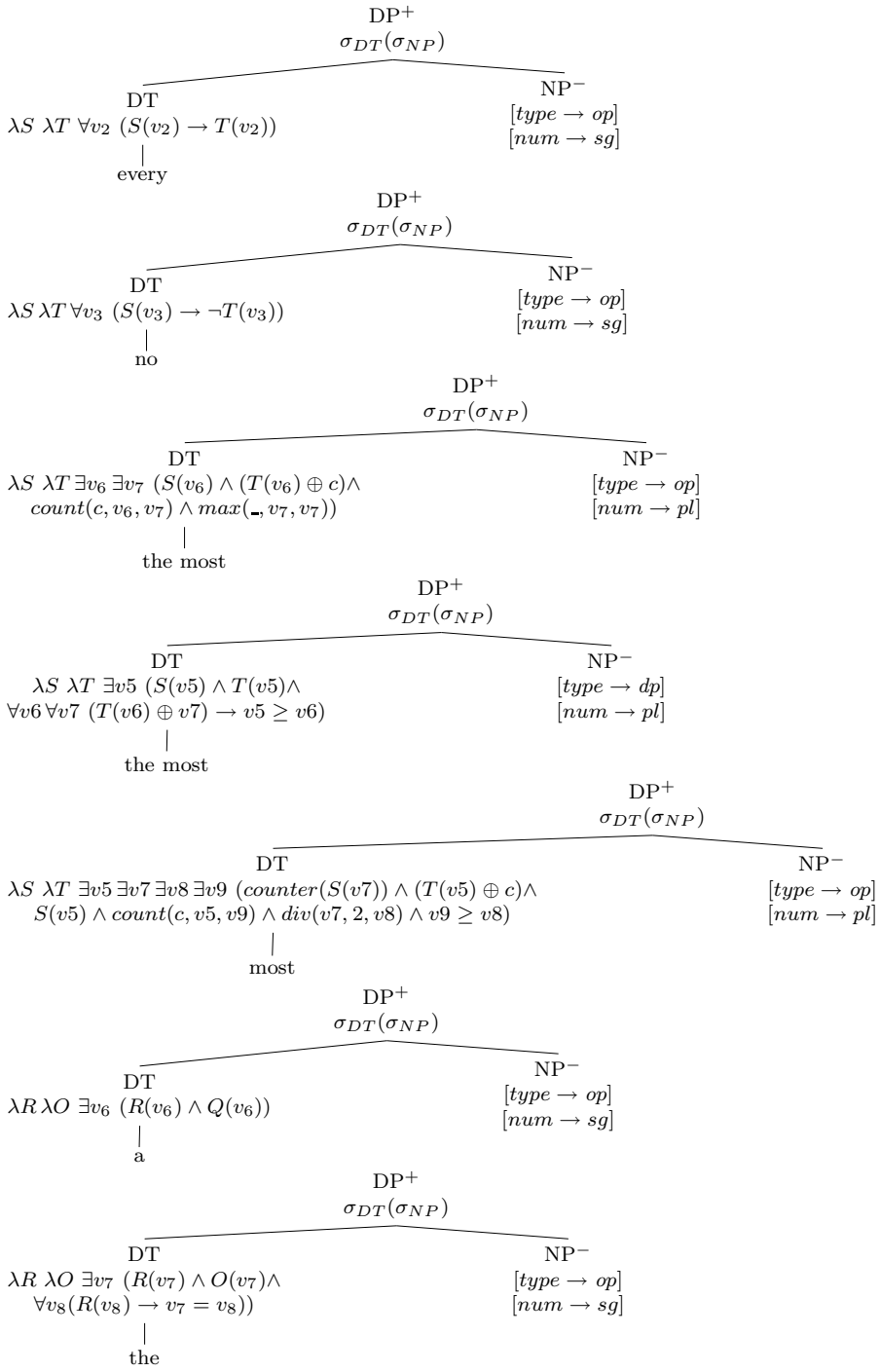
Chapter 9

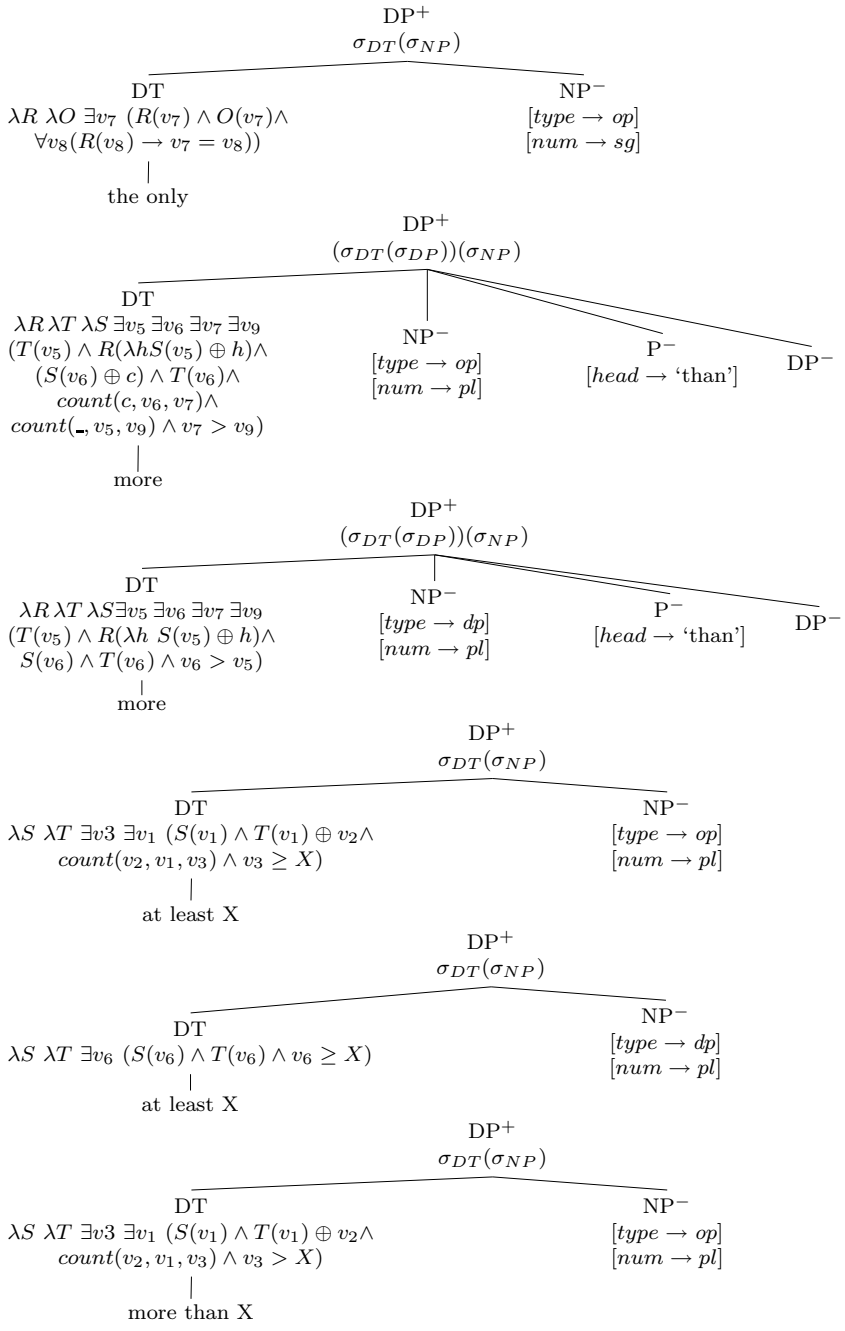
Appendix

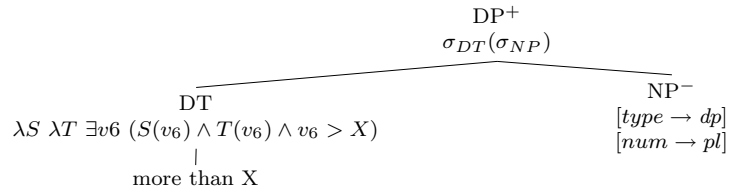
9.1 Determiners

This section gives the elementary trees for common determiners such as *how many, every, no, the most, most, a, the, the only, more than and at least.* Also other determiners are specified in the system's lexicon (e.g. *many, all, some*, etc.), but essentially they are similar to the determiners presented here, such that they are omitted. Many determiners come in two forms for object properties (op) and datatype properties (dp). Other F-Logic operators used in the semantic representations given below are *max, div* and *counter*. The operator $max(x, n, m)$ is evaluated such that m is the maximum of the values n grouped after x. As the count operator, if the x is omitted, we yield the absolute maximum of the n's. The $div(x, y, z)$ operator is evaluated to return the result (z) of dividing x by y. The operator $counter(S(x))$ counts the number of individuals fulfilling the monadic predicate S and binds this number to x.



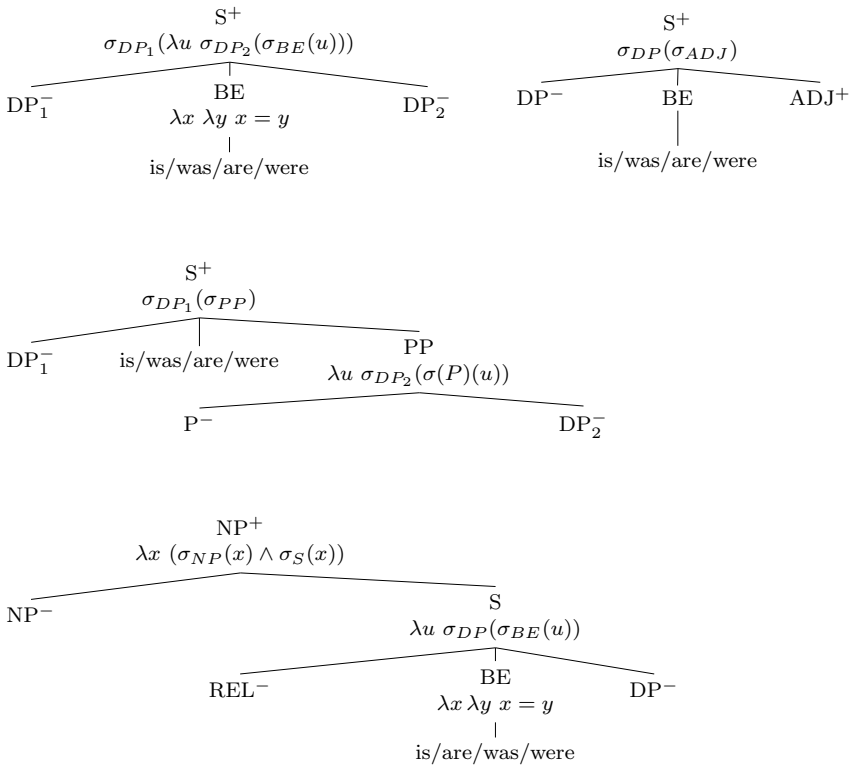

DP+
$\sigma_{DT}(\sigma_{NP})$
DT
$\lambda S\ \lambda T\ \forall v_2\ (S(v_2) \rightarrow T(v_2))$
every
NP−
$[type \rightarrow op]$
$[num \rightarrow sg]$
DP+
$\sigma_{DT}(\sigma_{NP})$
DT
$\lambda S\, \lambda T\, \forall v_3\ (S(v_3) \rightarrow \neg T(v_3))$
no
NP−
$[type \rightarrow op]$
$[num \rightarrow sg]$
DP+
$\sigma_{DT}(\sigma_{NP})$
DT
$\lambda S\ \lambda T\, \exists v_6\, \exists v_7\ (S(v_6) \wedge (T(v_6) \oplus c) \wedge$
$count(c, v_6, v_7) \wedge max(\_, v_7, v_7))$
the most
NP−
$[type \rightarrow op]$
$[num \rightarrow pl]$
DP+
$\sigma_{DT}(\sigma_{NP})$
DT
$\lambda S\ \lambda T\ \exists v5\ (S(v5) \wedge T(v5) \wedge$
$\forall v6\, \forall v7\ (T(v6) \oplus v7) \rightarrow v5 \geq v6)$
the most
NP−
$[type \rightarrow dp]$
$[num \rightarrow pl]$
DP+
$\sigma_{DT}(\sigma_{NP})$
DT
$\lambda S\ \lambda T\ \exists v5\, \exists v7\, \exists v8\, \exists v9\ (counter(S(v7)) \wedge (T(v5) \oplus c) \wedge$
$S(v5) \wedge count(c, v5, v9) \wedge div(v7, 2, v8) \wedge v9 \geq v8)$
most
NP−
$[type \rightarrow op]$
$[num \rightarrow pl]$
DP+
$\sigma_{DT}(\sigma_{NP})$
DT
$\lambda R\, \lambda O\ \exists v_6\ (R(v_6) \wedge Q(v_6))$
a
NP−
$[type \rightarrow op]$
$[num \rightarrow sg]$
DP+
$\sigma_{DT}(\sigma_{NP})$
DT
$\lambda R\ \lambda O\ \exists v_7\ (R(v_7) \wedge O(v_7) \wedge$
$\forall v_8(R(v_8) \rightarrow v_7 = v_8))$
the
NP−
$[type \rightarrow op]$
$[num \rightarrow sg]$


DP+
$\sigma_{DT}(\sigma_{NP})$
DT
$\lambda R\ \lambda O\ \exists v_7\ (R(v_7) \wedge O(v_7) \wedge \forall v_8(R(v_8) \rightarrow v_7 = v_8))$
the only
NP−
$[type \rightarrow op]$
$[num \rightarrow sg]$
DP+
$(\sigma_{DT}(\sigma_{DP}))(\sigma_{NP})$
DT
$\lambda R\,\lambda T\,\lambda S\,\exists v_5\,\exists v_6\,\exists v_7\,\exists v_9$
$(T(v_5) \wedge R(\lambda h S(v_5) \oplus h) \wedge$
$(S(v_6) \oplus c) \wedge T(v_6) \wedge$
$count(c, v_6, v_7) \wedge$
$count(\_, v_5, v_9) \wedge v_7 > v_9)$
more
NP−
$[type \rightarrow op]$
$[num \rightarrow pl]$
P−
$[head \rightarrow$ 'than']
DP−
DP+
$(\sigma_{DT}(\sigma_{DP}))(\sigma_{NP})$
DT
$\lambda R\,\lambda T\,\lambda S \exists v_5\,\exists v_6\,\exists v_7\,\exists v_9$
$(T(v_5) \wedge R(\lambda h\ S(v_5) \oplus h) \wedge$
$S(v_6) \wedge T(v_6) \wedge v_6 > v_5)$
more
NP−
$[type \rightarrow dp]$
$[num \rightarrow pl]$
P−
$[head \rightarrow$ 'than']
DP−
DP+
$\sigma_{DT}(\sigma_{NP})$
DT
$\lambda S\ \lambda T\ \exists v3\ \exists v_1\ (S(v_1) \wedge T(v_1) \oplus v_2 \wedge$
$count(v_2, v_1, v_3) \wedge v_3 \geq X)$
at least X
NP−
$[type \rightarrow op]$
$[num \rightarrow pl]$
DP+
$\sigma_{DT}(\sigma_{NP})$
DT
$\lambda S\ \lambda T\ \exists v_6\ (S(v_6) \wedge T(v_6) \wedge v_6 \geq X)$
at least X
NP−
$[type \rightarrow dp]$
$[num \rightarrow pl]$
DP+
$\sigma_{DT}(\sigma_{NP})$
DT
$\lambda S\ \lambda T\ \exists v3\ \exists v_1\ (S(v_1) \wedge T(v_1) \oplus v_2 \wedge$
$count(v_2, v_1, v_3) \wedge v_3 > X)$
more than X
NP−
$[type \rightarrow op]$
$[num \rightarrow pl]$



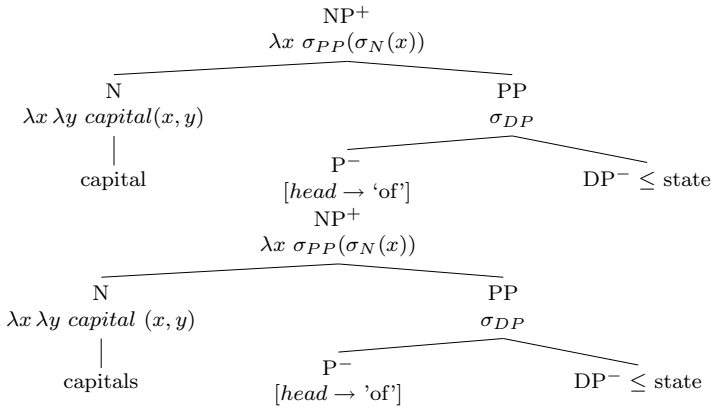
9.2 Copula

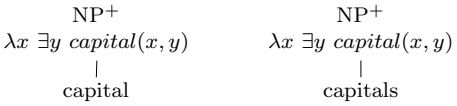
In this section we give the elementary trees for the copula verbs *'is'*, *'are'* and the past forms *'were'* and *'was'*.

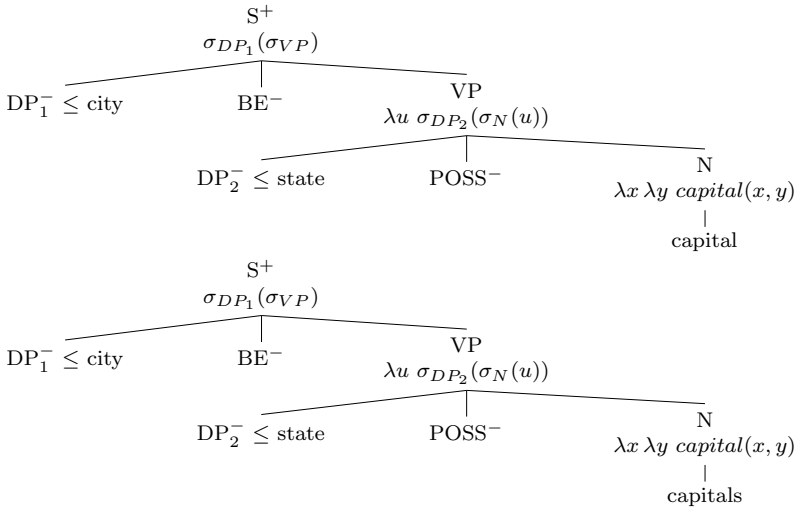


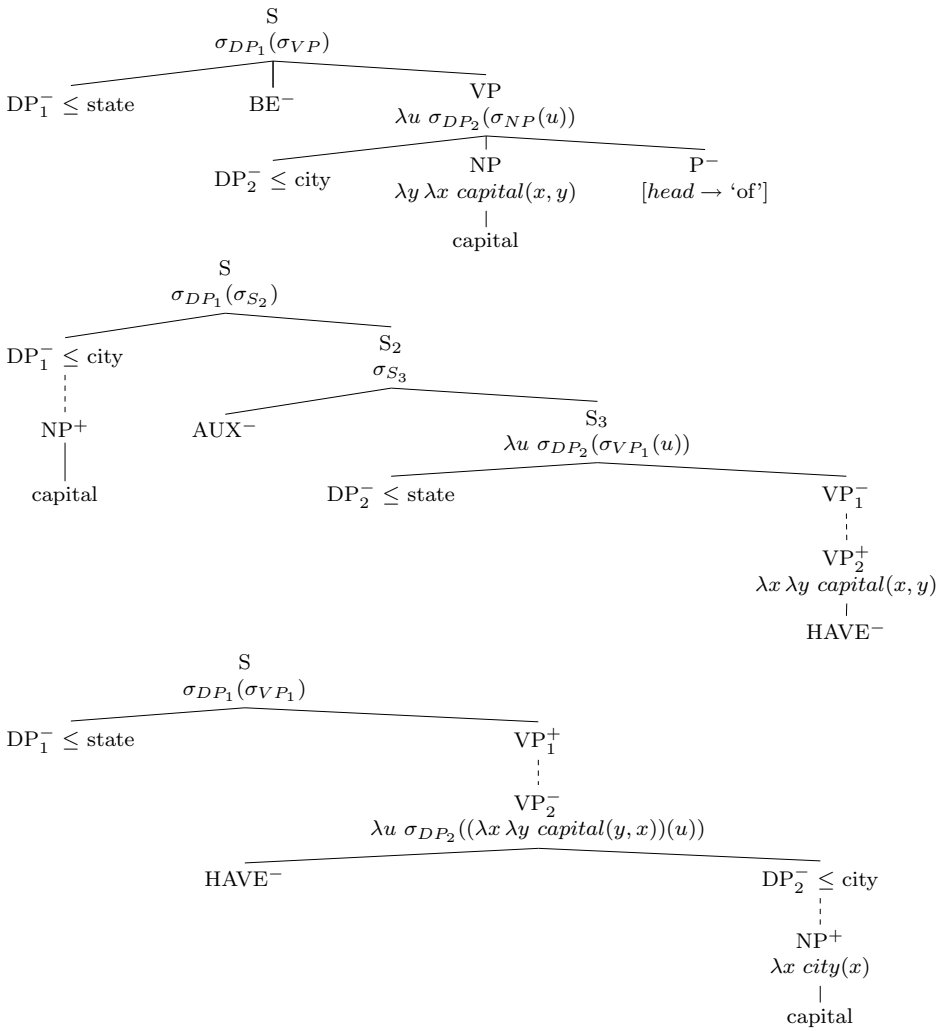
9.3 Noun phrase tree family

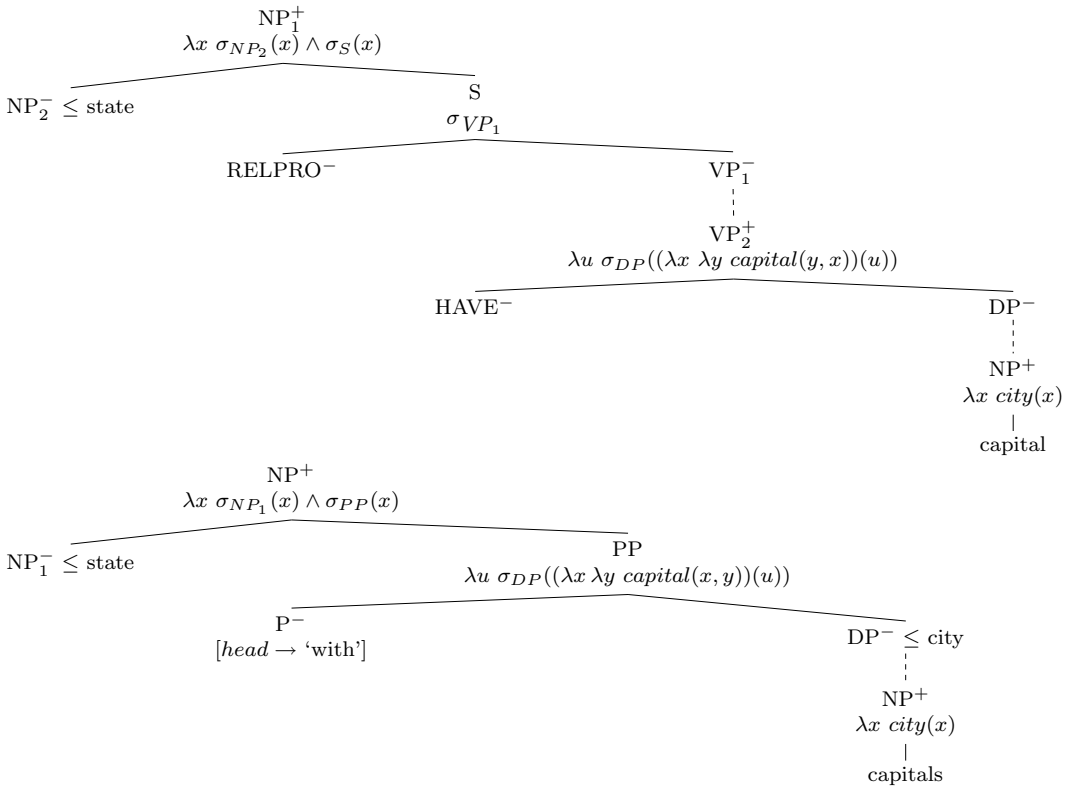
Here we give the tree family for a relational noun on the basis of the example for *capital*:


NP+
λx σPP(σN(x))
N
λx λy capital(x, y)
capital
PP
σDP
P−
[head → ‘of’]
DP− ≤ state
NP+
λx σPP(σN(x))
N
λx λy capital (x, y)
capitals
PP
σDP
P−
[head → ’of’]
DP− ≤ state


NP+
λx ∃y capital(x, y)
capital
NP+
λx ∃y capital(x, y)
capitals

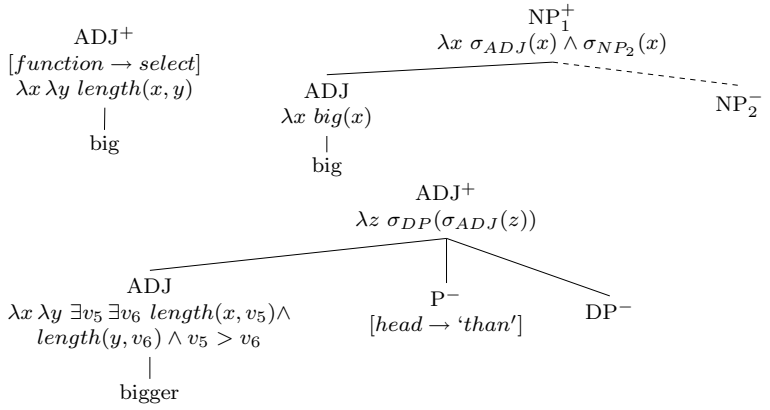

S+
σDP1(σVP)
DP1− ≤ city
BE−
VP
λu σDP2(σN(u))
DP2− ≤ state
POSS−
N
λx λy capital(x, y)
capital
S+
σDP1(σVP)
DP1− ≤ city
BE−
VP
λu σDP2(σN(u))
DP2− ≤ state
POSS−
N
λx λy capital(x, y)
capitals

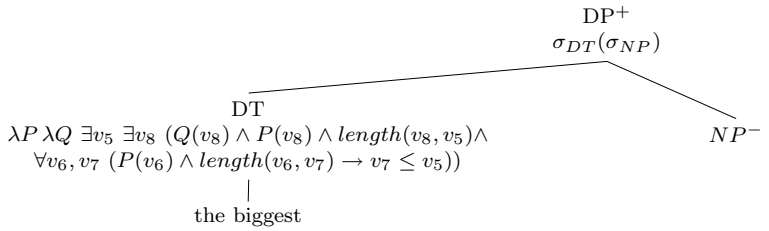

S
$\sigma_{DP_1}(\sigma_{VP})$
$\text{DP}_1^- \leq$ state
BE^-
VP
$\lambda u\ \sigma_{DP_2}(\sigma_{NP}(u))$
$\text{DP}_2^- \leq$ city
NP
$\lambda y\, \lambda x\ capital(x, y)$
capital
P^-
$[head \rightarrow$ 'of'$]$
S
$\sigma_{DP_1}(\sigma_{S_2})$
$\text{DP}_1^- \leq$ city
NP^+
capital
S_2
σ_{S_3}
AUX^-
S_3
$\lambda u\ \sigma_{DP_2}(\sigma_{VP_1}(u))$
$\text{DP}_2^- \leq$ state
VP_1^-
VP_2^+
$\lambda x\, \lambda y\ capital(x, y)$
HAVE^-
S
$\sigma_{DP_1}(\sigma_{VP_1})$
$\text{DP}_1^- \leq$ state
VP_1^+
VP_2^-
$\lambda u\ \sigma_{DP_2}((\lambda x\, \lambda y\ capital(y, x))(u))$
HAVE^-
$\text{DP}_2^- \leq$ city
NP^+
$\lambda x\ city(x)$
capital



9.4 Adjective Tree family

Tree family for adjectives with *'big'* as an example.




DP+
$\sigma_{DT}(\sigma_{NP})$
DT
$\lambda P\, \lambda Q\ \exists v_5\ \exists v_8\ (Q(v_8) \wedge P(v_8) \wedge length(v_8, v_5) \wedge$
$\forall v_6, v_7\ (P(v_6) \wedge length(v_6, v_7) \rightarrow v_7 \leq v_5))$
the biggest
NP^-

Bibliography

[Androutsopoulos et al., 1995] Androutsopoulos, I., Ritchie, G., and Thanisch, P. (1995). Natural language interfaces to databases–an introduction. *Journal of Language Engineering*, 1(1):29–81.

[Ankolekar et al., 2006] Ankolekar, A., Buitelaar, P., Cimiano, P., Hitzler, P., Kiesel, M., Krötzsch, M., Lewen, H., Neumann, G., Sintek, M., Tserendorj, T., and Studer, R. (2006). Smartweb: Mobile access to the semantic web. In *Proceedings of the ISWC 2006 Poster and Demo Session*.

[Baeza-Yates and Ribeiro-Neto, 1999] Baeza-Yates, R. and Ribeiro-Neto, B. (1999). *Modern Information Retrieval*. Addison-Wesley.

[Baker et al., 1998] Baker, C., Fillmore, C., and Lowe, J. (1998). The Berkeley FrameNet project. In *Proceedings of the International Conference on Computational Linguistics and the Annual Meeting of the Association for Computational Linguistics (COLING-ACL)*.

[Bangalore and Joshi, 1999] Bangalore, S. and Joshi, A. (1999). Supertagging: An approach to almost parsing. *Computational Linguistics*, 25(2):237–265.

[Bates, 1989] Bates, M. (1989). Rapid porting of the parlance natural language interface. In *Proceedings of the Workshop on Speech and Natural Language*, pages 83–88.

[Bechhofer et al., 2004] Bechhofer, S., van Harmelen, F., Hendler, J., Horrocks, I., McGuinees, D., Patel-Schneider, P., and Stein, L. (2004). OWL Web Ontology Language Reference. http://www.w3.org/TR/owl-ref.

[Bernstein et al., 2005] Bernstein, A., Kaufmann, E., Göhring, A., and Kiefer, C. (2005). Querying ontologies: A controlled english interface for end-users. In *Proceedings of the 4th International Semantic Web Conference (ISWC)*, pages 112–126.

[Blackburn and Bos, 2005] Blackburn, P. and Bos, J. (2005). *Representation and Inference for Natural Language - A First Course in Computational Semantics*. CSLI Publications.

[Brickley and Guha, 2004] Brickley, D. and Guha, R. (2004). RDF Vocabulary Description Language 1.0: RDF Schema. W3C Recommendation. available at http://www.w3.org/TR/rdf-schema/.

[Broekstra et al., 2002] Broekstra, J., Kampman, A., and van Harmelen, F. (2002). Sesame: A generic architecture for storing and querying rdf and rdf schema. In *Proceedings of the International Semantic Web Conference (IWSC)*, pages 54–68.

[Cimiano, 2003] Cimiano, P. (2003). Translating wh-questions into F-Logic queries. In Bernardi, R. and Moortgat, M., editors, *Proceedings of the CoLogNET-ElsNET Workshop on Questions and Answers: Theoretical and Applied Perspectives*, pages 130–137.

[Cimiano, 2004] Cimiano, P. (2004). ORAKEL: A Natural Language Interface to an F-Logic Knowledge Base. In *Proceedings of the 9th International Conference on Applications of Natural Language to Information Systems (NLDB)*, pages 401–406.

[Cimiano et al., 2007] Cimiano, P., Haase, P., and Heizmann, J. (2007). Porting natural language interfaces between domains – a case study with the ORAKEL system –. In *Proceedings of the International Conference on Intelligent User Interfaces (IUI)*, pages 180–189.

[Cimiano et al., 2006] Cimiano, P., Haase, P., Sure, Y., Völker, J., and Wang, Y. (2006). Question answering on top of the BT digital library. In *Proceedings of the World Wide Web conference (WWW)*, pages 861–862.

[Cimiano and Reyle, 2003] Cimiano, P. and Reyle, U. (2003). Ontology-based semantic construction, underspecification and disambiguation. In *Proceedings of the Prospects and Advances in the Syntax-Semantic Interface Workshop*, pages 33–38.

[Cimiano and Reyle, 2006] Cimiano, P. and Reyle, U. (2006). Towards foundational semantics - ontological semantics revisited -. In *Proceedings of the International Conference on Formal Ontology in Information Systems (FOIS)*, volume 150, pages 51–62. IOS Press.

[Copestake et al., 2006] Copestake, A., Flickinger, D., Pollard, C., and Sag, I. (2006). Minimal recursion semantics: An introduction. *Research on Language and Computation*, (3):281–332.

[Copestake and Jones, 1989] Copestake, A. and Jones, K. S. (1989). Natural language interfaces to databases. *Knowledge Engineering Review*, 5(4):225–249. Special Issue on the Applications of Natural Language Processing Techniques.

[Cunningham et al., 1997] Cunningham, H., Humphreys, K., Gaizauskas, R., and Wilks, Y. (1997). GATE - a General Architecture for Text Engineering. In *Proceedings of Applied Natural Language Processing (ANLP)*, pages 29–30.

[Decker et al., 1999] Decker, S., Erdmann, M., Fensel, D., and Studer, R. (1999). Ontobroker: Ontology Based Access to Distributed and Semi-Structured Information. In *Database Semantics: Semantic Issues in Multimedia Systems*, pages 351–369. Kluwer.

[Dekker, 1993] Dekker, P. (1993). Existencial disclosure. *Linguistics and Philosophy*, (16):561–587.

[E. Bozsak et al., 2002] E. Bozsak et al. (2002). KAON - Towards a large scale Semantic Web. In *Proceedings of the Third International Conference on E-Commerce and Web Technologies (EC-Web)*. Springer Lecture Notes in Computer Science.

[Fellbaum, 1998] Fellbaum, C. (1998). *WordNet, an electronic lexical database*. MIT Press.

[Frank et al., 2007] Frank, A., Krieger, H.-U., Xu, F., Uszkoreit, H., Crysmann, B., Jörg, B., and Schäfer, U. (2007). Question answering from structured knowledge sources. *Journal of Applied Logic, Special Issue on Questions and Answers: Theoretical and Applied Perspectives*, 5(1):20–48.

[Fuchs et al., 2006] Fuchs, N., Kaljurand, K., and Schneider, G. (2006). Attempto controlled english meets the challenges of knowledge representation, reasoning, interoperability and user interfaces. In *Proceedings of the International Conference of the Florida Artificial Intelligence Research Society (FLAIRS)*.

[Grosz et al., 1987] Grosz, B., Appelt, D., Martin, P., and Pereira, F. (1987). Team: An experiment in the design of transportable natural language interfaces. *Artificial Intelligence*, 32:173–243.

[Haase et al., 2004] Haase, P., Broekstra, J., Eberhart, A., and Volz, R. (2004). A comparison of RDF query languages. In *Proceedings of the 3rd International Semantic Web Conference (ISWC)*.

[Hendrix et al., 1978] Hendrix, G., Sacerdoti, E., Sagalowicz, D., and Slocum, J. (1978). Developing a natural language interface to complex data. *ACM Transactions on Database Systems*, 3(2):105–147.

[Joshi and Schabes, 1997] Joshi, A. and Schabes, Y. (1997). Tree-adjoining grammars. In *Handbook of Formal Languages*, volume 3, pages 69–124. Springer.

[Kamp and Reyle, 1993] Kamp, H. and Reyle, U. (1993). *From Discourse to Logic*. Kluwer.

[Kifer et al., 1995] Kifer, M., Lausen, G., and Wu, J. (1995). Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42:741–843.

[Lopez and Motta, 2004] Lopez, V. and Motta, E. (2004). Aqualog: An ontology-portable question answering system for the semantic web. In *Proceedings of the International Conference on Natural Language for Information Systems (NLDB)*, pages 89–102.

[Masolo et al., 2003] Masolo, C., Borgo, S., Gangemi, A., Guarino, N., and Oltramari, A. (2003). Ontology library (final). WonderWeb deliverable D18.

[McGuiness and van Harmelen, 2004] McGuiness, D. and van Harmelen, F. (2004). OWL Web Ontology Language Overview. W3C Recommendation. available at http://www.w3.org/TR/owl-features/.

[Minock, 2005] Minock, M. (2005). A phrasal approach to natural language interfaces over databases. Technical Report UMINF-05.09, University of UMEA, Department of Computer Science.

[Montague, 1974] Montague, R. (1974). On the proper treatment of quantification in ordinary english. In Thomason, R. H., editor, *Formal Philosophy: Selected Papers of Richard Montague*, pages 247–270.

[Muskens, 2001] Muskens, R. (2001). Talking about trees and truth-conditions. *Journal of Logic, Language and Information*, 10(4):417–455.

[Nirenburg and Raskin, 2004] Nirenburg, S. and Raskin, V. (2004). *Ontological Semantics*. MIT Press.

[Ogden and Bernick, 1996] Ogden, W. and Bernick, P. (1996). Using natural language interfaces. In Helander, M., editor, *Handbook of Human-Computer Interaction*. Elsevier.

[Pease et al., 2002] Pease, A., I.Niles, and Li, J. (2002). The suggested upper merged ontology: A large ontology for the semantic web and its applications. In *Working Notes of the AAAI-2002 Workshop on Ontologies and the Semantic Web*.

[Popescu et al., 2003] Popescu, A., Etzioni, O., and Kautz, H. (2003). Towards a theory of natural language interfaces to databases. In *Proceedings of the International Conference on Intelligent User Interfaces (IUI'03)*, pages 149–157.

[Prud'hommeaux and Seaborne, 2006] Prud'hommeaux, E. and Seaborne, A. (2006). Sparql query language for rdf. W3C Working Draft 4. available at http://www.w3.org/TR/rdf-sparql-query/.

[Rose et al., 2005] Rose, C., Pai, C., and Arguello, J. (2005). Enabling non-linguists to author advanced conversational interfaces easily. In *Proceedings of the International Conference of the Florida Artificial Intelligence Research Society (FLAIRS)*, pages 572–577.

[Schabes et al., 1988] Schabes, Y., Abeille, A., and Joshi, A. (1988). Parsing strategies with 'lexicalized' grammars: application to tree adjoining grammars. In *Proceedings of the International Conference on Computational Linguistics (COLING'88)*, pages 578–583.

[Schabes and Joshi, 1988] Schabes, Y. and Joshi, A. (1988). An earley-type parsing algorithm for tree adjoining grammars. Technical Report MS-CIS-88-36 / LINC LAB 113, University of Pennsylvania.

[Schmid, 1994] Schmid, H. (1994). Probabilistic part-of-speech tagging using decision trees. In *Proceedings of the International Conference on New Methods in Language Processing*.

[Shneiderman and Plaisant, 2005] Shneiderman, B. and Plaisant, C. (2005). *Designing the User Interface*. Pearson/Addison-Wesley.

[Thompson and Thompson, 1985] Thompson, B. and Thompson, F. (1985). ASK is transportable in half a dozen ways. *ACM Transactions on Office Information Systems*, 3(2):185–203.

[Thompson et al., 1997] Thompson, C., Mooney, R., and Tang, L. (1997). Learning to parse natural language database queries into logical form. In *Proceedings of the Workshop on Automata Induction, Grammatical Inference and Language Acquisition*.

[Warren and Alsmeyer, 2005] Warren, P. and Alsmeyer, D. (2005). Applying semantic technology to a digital library: a case study. *Library Management*, 26(4/5):190–195. Special Issue: Semantic Web.

[Zadeh, 1975] Zadeh, L. (1975). The concept of a linguistic variable and its application to approximate reasoning. *Information Sciences*, 8-9.