Lambda Calculus on Dependency Structures for a Wide-Coverage Grammar of Esperanto

Making Deep Semantics More Robust

Thesis submitted for the degree of Bachelor of Arts

Seminar für Sprachwissenschaft
Eberhard Karls Universität Tübingen

Author:
Johannes Dellert
johannes.dellert@gmx.de

Course:
Computational Semantics

Supervisor:
Dr. Frank Richter

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

J. Dellert

(Johannes Dellert)

Contents

1 Introduction

In recent years, the explosion of computing power caused by the continued progress of information technology has finally made large-scale automatic reasoning possible. In the wake of these developments, deep computational semantics has become a computationally viable option for natural language processing (NLP). Experimental systems are developed at many places, as reported e.g. in Bos (2001) and Curran et al. (2007). Further rapid progress in the field is to be expected during the next decade.

Blackburn and Bos (2005), henceforth BB05, provide code and documentation for a very illustrative demo system called CURT (Clever Use of Reasoning Tools) that is able to convert a small fragment of English into first-order formulae. CURT uses automatic reasoning tools to detect a few types of inconsistency and redundancy in natural language input as well as to answer natural language questions about a situation the user describes.

In this thesis, I endeavour to construct an Esperanto version of CURT, which by a completely analogous acronym I call LARA (Lerta Apliko de Rezoniloj Aŭtomataj). The syntactic coverage of LARA comes close to that of CURT, and the particularly nice features of Esperanto morphology that make lexical entries for open word classes almost unnecessary allow me to provide for a much wider lexical coverage.

Large parts of the CURT system could directly be taken over into LARA. However, where CURT uses nothing more sophisticated than Prolog's definite clause grammar (DCG) mechanism for syntactic parsing, LARA's semantic computations are guided by a dependency grammar (DG). To produce dependency structures, I integrated my Dependency Parser with User-definable Relation Constraints (DPURC) into the system.

I justify my choice to use a DG in chapter 2, where I also explain why I consider it worthwile to experiment with Esperanto in computational semantics. As I will also show there, applying the lambda calculus to dependency structures leads to a few interesting problems.

In chapter 3, I give a first overview of the system architecture and explain the interaction of the various components that are later described in more detail.

In chapter 4, I present and discuss my approach to applying lambda calculus to dependency structures: a simple constraint system that allows to reduce the combinatorical complexity of the task and allows to state rules for the semantic processing of dependency structures without too much redundancy.

One advantage of the system is that it can easily be adapted to allow for robust partial interpretation of complicated syntactic structures. I discuss some theoretical aspects of this idea and a few previous approaches to it in chapter 5.

Chapter 6 then introduces my Prolog implementation of LARA, where the results from the two preceding chapters are put into practice and demonstrated in a small example dialogue.

Since the implementation heavily relies on components from CURT, familiarity with Prolog as well as basic knowledge of the internal workings of that system are highly recommended to fully understand my documentation of LARA. Furthermore, to be able to understand the details of semantic processing, previous exposure to some form of Montague-style formal semantics is necessary.

The complete code for LARA that is needed in addition to the CURT system is contained in the appendix, except for the syntactic dependency parser DPURC, which is documented and distributed separately in Dellert (forthcoming).

2 Esperanto and Dependency Grammar

2.1 Esperanto

Esperanto is an international auxiliary language invented by Zamenhof [under the pseudonym Ėsperanto] (1887). While its lexical items are mainly borrowed from major European languages, its derivational and inflectional morphology is a lot more systematic and productive than in those, making the language much more similar to agglutinative languages such as Turkish or Hungarian in that respect.

Throughout the 20th century, Esperanto has been the most widespread constructed language. There is an active global community of speakers, with number estimates between 100.000 and as many as two million speakers. Despite its artificial nature, its speakers have turned the language into a rich and effective means of communication, with many traits otherwise only found in natural languages, such as creative neologisms, idiomatic expressions, and stylistic variance from literary to colloquial usage. Large text corpora in Esperanto are freely available on the web, with the Esperanto Wikipedia (more than 100.000 articles) as one of the largest.

2.1.1 Why use it as a test case for NLP?

The unique character of Esperanto has been attracting researchers in NLP for a long time. For many experiments, Esperanto offers a close to ideal compromise between regularity (as in a formal language) and expressiveness (as in a natural language). This hybrid character has predestined Esperanto to play a role e.g. as an interlingua for machine translation. Schubert (1988) explains in detail why (a somewhat modified version of) Esperanto was chosen to be the interlingua in the multilingual translation system DLT. The reasons he mentions certainly also apply to other areas of high-level NLP, where ways to treat semantics or pragmatics have to be tested without spending too much time on low-level issues that usually cause a good deal of the work needed to build experimental systems.

2.1.2 Morphology and lexicon made simple

The key advantage of Esperanto as a test case for high-level NLP is its completely regular morphology that makes the otherwise complex tasks of part-of-speech tagging and morphological analysis almost trivial. Esperanto morphology encodes open word classes with signal vowels, e.g. adjectives with the vowel *a* and nouns with the vowel *o*. These endings allow to infer a good deal of lexical information during morphological analysis. This feature of the language causes a much less urgent need for comprehensive dictionaries in the context of many NLP applications. However, since the language has developed into a full replacement for natural languages in all situations, all the aspects of pragmatics and semantics that NLP wants to address are present in Esperanto as much as in any natural language. These advantages have allowed me to address the high-level issues discussed here without any need to spend much time on preparatory low-level work.

2.2 Dependency Grammar

Dependency Grammar (DG) is a grammar formalism that builds on directed dependency arcs between words in a sentence. Each of these arcs links a *dependent* to a *head* or *governor* and is labeled with the role of the dependent in relation to the head. The words of a sentence and the dependency links between them form a graph structure, the *dependency tree*. Such a tree only contains nodes for each word, not for phrases or sentences. Dependency trees are usually headed by the main verb of a sentence, and the heads of all the arguments of the verb are directly linked to it. This makes dependency trees less deep than typical phrase structure trees (see Figure 1 for an example), and makes it difficult to e.g. define verbal phrases as constituents.

To a certain degree, a constituent structure can be inferred from a dependency structure (as long

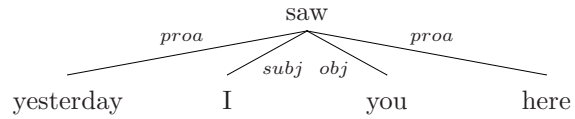

Figure 1: dependency structure for *yesterday I saw you here*

as there are no crossing dependencies, i.e. as long as the dependency structure is indeed a tree) by defining that each node forms a constituent together with the subtree headed by it. Dependency grammars that exclude discontinuous constituents by disallowing crossing dependencies are called *projective*. Projective DGs can easily be shown to equal context-free grammars in expressivity. However, by allowing non-projectivity, it is possible to go beyond this boundary of expressiveness.

The crucial difference between phrase structure grammars and dependency grammars is the form of their rules: Where phrase structure grammars rely on fixed phrase patterns to describe possible structures, each dependency grammar rule only defines one possible dependency link between a head and a dependent without imposing any restrictions on the other dependents of the head. As a consequence, there is no concept of linear precedence inherent to DGs, which means that any linear order of dependents can be recognized by default, making DGs very efficient for describing free word-order languages. On the other hand, it is hard to restrain word order using DGs, which easily leads to severe overgeneration for languages with fixed word order.

Another problem of DGs is that it is difficult to enforce the presence of dependents, e.g. to state that a finite verb must have a subject. It is even more difficult to let the presence of certain dependents enforce the presence of others, e.g. to state that a coordinating conjunction that has one noun phrase as a dependent must have another one. The variability of structures recognized by default makes dependency grammars better at finding structures for unexpected input, but also easily leads to incomplete or otherwise invalid structures.

2.2.1 Why is it useful for Esperanto?

Informal variants of Dependency Grammar have more than two millennia of tradition in linguistics. Dependency Grammar was used by Latin and Greek grammarians to describe the free word-order syntax of their languages. After the advent of modern linguistics, research on DG continued mainly in the context of Slavic grammar traditions. Dependency syntax tends to be more popular for languages with free word order due to the fact that in a classical context-free phrase stucture grammar, free word order has to be emulated by stating rules for each of the possible subconstituent orderings. A very simple way of determining the usefulness of dependency structures for Esperanto syntax is thus to observe the degree of word order variability present in the language. The following example gives a good impression of the fact that Esperanto word order is indeed rather free:

(1) (a) *La* *granda* *virino* *vidis* *malgrandan* *viron.*
the tall-NOM woman-NOM see-PAST small-ACC man-ACC
'The tall woman saw a small man'

(b) Malgrandan viron vidis la granda virino.

(c) Vidis la virino granda viron malgrandan.

(d) Vidis viron malgrandan la granda virino.

These were just four of many completely valid different word orders for a relatively short sentence. This alone should be sufficient proof that dependency structures are indeed a good choice for describing the syntax of Esperanto.

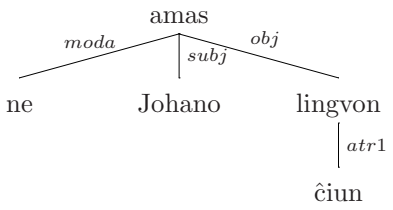

Figure 2: dependency structure for *Johano ne amas ĉiun lingvon*

Another reason for using dependency structures to process Esperanto syntax in the context of LARA was that Schubert (1989) presents a readily usable wide-coverage dependency grammar for Esperanto, a fragment of which formed the basis for the grammar used by LARA's syntactic parsing component.

2.2.2 Challenges for lambda calculus on dependency structures

One of the goals of this thesis is to explore ways of deriving logical forms for sentences that are analysed syntactically within the framework of dependency grammar. One of the more obvious approaches is to try to apply a well-established semantic formalism such as the lambda calculus to dependency structures. In this manner, one could make direct use of a lot of semantic literature that uses the lambda calculus as a formal device. Consider, however, the sentence

(2) *Johano* *ne* *amas* *ĉiun* *lingvon.*
John-NOM not like-PRES every-ACC language-ACC
'John does not like every language.'

to which DPURC assigns the dependency structure in Figure 2.

This rather simple structure already features several potential challenges for a direct application of lambda calculus. Most importantly, the structures are not binary branching, a property of typical phrase structure trees that made functional application particularly easy because in principle it only left one choice at each node, namely whether to apply the semantic representation of the left child to the right child or vice versa. In the case of dependency structures where one node can easily have more than two children, the situation gets much more complicated. It is not only unclear which representations must be applied to which, but also in what order the functional applications must occur. In addition, contrary to the situation in a typical phrase structure tree, not all the semantic information is already present in the leaves. The internal nodes also represent words with meaning, which means that they also have to enter the derivation of the logical form at some point.

A careful consideration of the dependency structure together with the semantic representations for its nodes (Figure 3) reveals that the functional applications in Figure 4 are necessary to derive the desired logical form for the sentence. The semantic brackets are used here to denote the mapping from natural language sentences to logical forms, not the model-theoretic interpretation of those forms. This means that the functional applications are only considered means of manipulating logical symbols here, which closely mirrors the perspective of a computer system that performs computations in lambda calculus. While in a typical phrase structure tree, at each node we only have to compute one functional application, here we have one node that requires at least three applications in a specific order. The negation must be applied to the verb, the result to the object, and then the subject to the result.

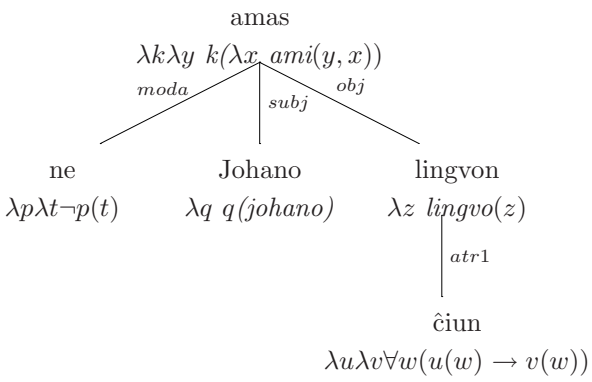

Figure 3: dependency structure for *Johano ne amas ĉiun lingvon* with semantics on nodes

⟦ ĉiun lingvon ⟧ = ⟦ ĉiu ⟧(⟦ lingvo ⟧)
⟦ ne amas ⟧ = ⟦ ne ⟧(⟦ ami ⟧)
⟦ ne amas ĉiun lingvon⟧ = ⟦ ne amas ⟧(⟦ ĉiun lingvon ⟧)
⟦ Johano ne amas ĉiun lingvon⟧ = ⟦ Johano ⟧(⟦ ne amas ĉiun lingvon ⟧)

Figure 4: functional applications necessary to compute the semantics of sentence 2

To automatically process dependency structures in such complex ways, one could write rules that explicitly enumerate all possible configurations of dependencies at each node and define the processing order and directionality of all the functional applications in each of those cases. However, for any grammar fragment that goes beyond toy coverage this would lead to severe problems with redundancies similar to those seen in phrase structure grammars that describe free word order languages. It thus seems recommendable to devise mechanisms for treating dependency structures in a more efficient and natural manner.

2.2.3 Existing approaches to deep semantics using dependency grammar

Given the mentioned difficulties, one might think it wise not to use lambda calculus as a formal device for computing deep semantics on dependency structures. Traditionally, dependency grammar was mostly used for shallow semantics that is easier to combine with robust parsing. For that reason, not very much has been published about deep semantics with dependency grammar yet. The predominant tradition in the area are semantic networks (see e.g. Sowa (1984) for details, and Robaldo (2007) for an overview of recent research in the field). Semantic networks make use of the fact that many semantic relations we are interested in are already mirrored in dependency structures a lot more clearly than in phrase structure trees, such that the computation of many semantic relations amounts to not much more than relabeling dependency links. However, simple (i.e. relational) semantic networks have less expressive power than first-order predicate logic, which prevents any reasonable interpretation of quantifiers and makes relational networks too weak for logical inferencing. Only by adding an analogy to variable binding and quantifier scope in the form of overlapping contexts, semantic nets (which are then called propositional networks) achieve the expressiveness of first order logic. Unfortunately, however, equal expressiveness does not necessarily mean that such structures can easily be converted into first-order formulae. As a consequence, inferencing and automatic reasoning on semantic nets are a lot less straightforward than on logical forms derived via the lambda calculus. Given this and the fact that mainstream research in deep semantics usually takes place in some variant of lambda calculus, it is certainly worthwile to pursue the idea of lambda calculus on dependency structures further.

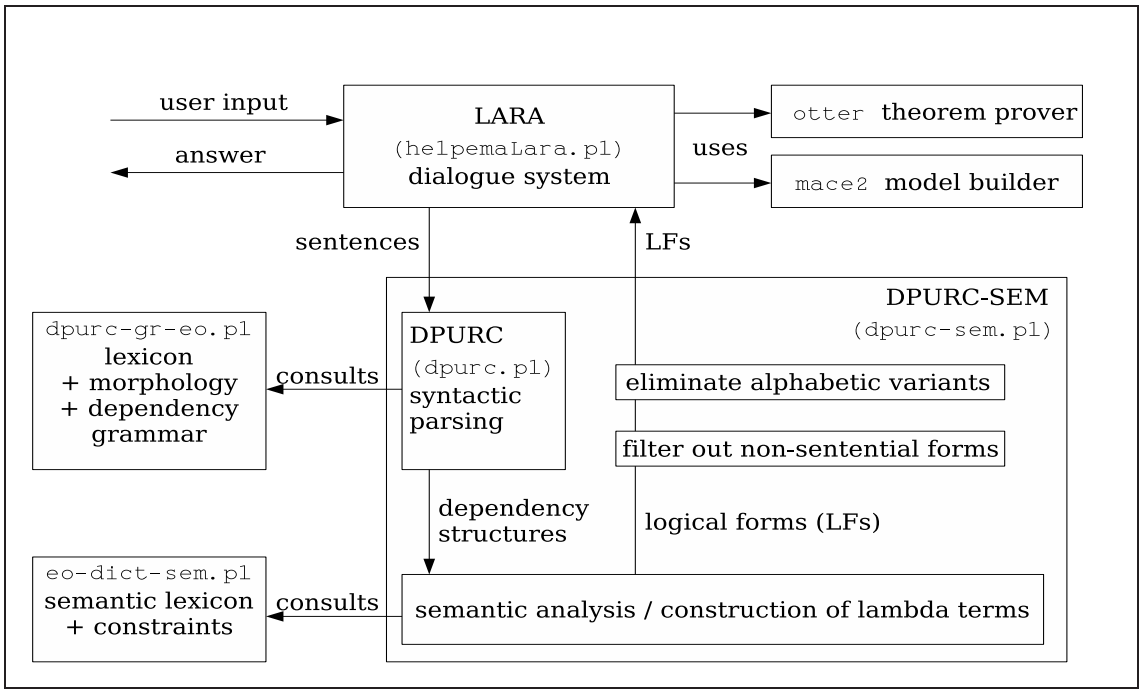

Figure 5: LARA's system architecture

3 Architecture of the LARA system

In order to understand the details of semantic processing in LARA, it is useful to have a clear picture of the different components of the system. Figure 5 shows a graph of the system's architecture.

LARA is in essence a simple dialogue system that allows the user to input sentences which it then interprets model-theoretically. The conversion of input sentences into first-order logical forms that LARA can understand takes place in the DPURC-SEM component. Semantic processing in DPURC-SEM is accomplished in two steps: DPURC-SEM first hands on the sentences to the syntactic dependency parser DPURC to retrieve possible dependency structures. These structures are then used by DPURC-SEM to compute interpretations for the sentences. This computation will be the focus of discussion in the next sections.

To determine how to process Esperanto sentences, both the syntactic and the semantic processing component rely on lexical entries and rules or constraints that are specified in external files. For DPURC, the necessary morphological and syntactic information is contained in `dpurc-gr-eo.pl`. The lexical entries and application constraints for DPURC-SEM are contained in `eo-dict-sem.pl`. More information on structure and content of those two files can be found in sections 6.3 and 6.4.

For reasons discussed in section 6.5, the set of logical forms returned by the semantic processing component contains many non-sentential or syntactically incomplete forms. These forms must be filtered out in a post-processing step. The remaining set usually contains some logical forms in many alphabetic variants, which are eliminated in a second post-processing step.

The surviving forms are sentences of first-order logic which DPURC-SEM hands back to LARA as the possible interpretations of the input sentence. LARA then uses external automated reasoning tools to check whether these interpretations are consistent with the previous input and whether

they contain information that could not have been inferred from the discourse. If one of these two situations occurs, LARA acts accordingly by reporting this to the user. To achieve this ability, LARA uses a model builder to construct models of the knowledge inferred from user input. LARA is then also able to answer some kinds of natural-language questions about the situation the user described by consulting those models.

4 Applying Lambda Calculus to Dependency Structures

As we have seen, the main problem for performing traditional semantic lambda calculus on dependency structures is that those structures often contain heads with more than two dependents. It is not possible for the computer to tell the correct order of functional applications to derive the semantics of such a head. Moreover, the correct directionality of functional applications is not known either. One could argue that this problem also occurs during lambda calculus on phrase structure trees, but can be alleviated by simply defining the correct functional application along with each syntactic rewriting rule. However, in a formalism like dependency grammar that is not based on rewriting rules, the possible dependents to a word are not encoded in a single rule, but by a multitude of rules defining one possible dependency link each. Therefore, the necessary one-to-one mapping between rule and resulting structure cannot be established. Listing rules for each possible configuration of heads and dependents would lead to a lot of redundancy and we would risk losing the ability to interpret unexpected structures.

4.1 A Simple Constraint-Based Approach

The most primitive approach to solving these problems would be to simply try out all alternatives at each node, i.e. all combinatorically possible combinations of directional applications. One could then simply collect all the resulting lambda expressions and filter duplicates as well as non-sentential forms.

It is obvious that this would work, but since we are working with an untyped lambda calculus, it would be impossible to confine the search space early on, resulting in an undirected search that would result in unacceptably bad computational behaviour. To illustrate the severity of this problem, consider once more the dependency structure in Figure 2. At the root node, we have 3! different possible orders for the dependents, where each order would result in three functional applications with 3^2 directional variants. The quantifier in this structure contributes one additional choice of directionality for one functional application. As a result, we would have to try $3! * 3^2 * 2 = 108$ different ways of processing the dependency structure, with only one valid logical form that would have to be sieved out by elimination of duplicates and non-sentential forms. It is fairly obvious that the runtime of such an algorithm becomes unacceptable very soon.

The basic idea to alleviate this problem is to devise and enforce constraints on possible functional applications. Assume, for example, that we must compute the semantics of a noun phrase consisting of a noun and an article, where the noun's semantic lexical entry is something like $\lambda x\ human\ (x)$ and the article's entry something like $\lambda u \lambda v \exists x(u(x) \wedge v(x))$. From our knowledge and experience of how semantical derivations work, we can tell that in this case, the article's meaning must be applied to the noun's meaning, because otherwise we get a wrong semantic representation for the whole noun phrase. It would be desirable to impose a constraint on the computation that forbids application of a noun to its article and enforces application in the other direction. I will call such constraints *directionality constraints*.

Directionality constraints are not the only type of constraints that will be useful. Consider another noun phrase that apart from the noun and the article contains an adjective. The dependency structure for such a noun phrase could look like the one in Figure 6.

In this case, directionality constraints could tell us that the article and the adjective must both

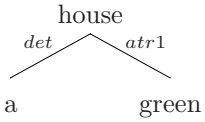

Figure 6: dependency structure for the English noun phrase *athe green house*

be applied to the noun. This leaves open the order of the two applications, such that the system would have to try out two alternatives: First apply the article to the noun, and then the adjective to the result, or apply the adjective first, and then the article. Again experience tells us that only the second alternative will lead to the correct semantics for the noun phrase. It is thus certainly worthwile to include a possibility to impose constraints on the order in which applications occur. In our case, such a constraint would state that an attributive adjective has to be processed together with its head noun before the noun's article comes in. I will call such constraints *precedence constraints*.

As will become apparent in the LARA system, the interaction of these two types of constraints allows to guide the derivation of logical forms in very useful ways. However, some problems and sources of inefficiency remain; for instance, the approach sometimes unnecessarily leads to spurious readings.

Spurious readings are encountered e.g. if a head has two incoming dependency links with identical labels. This problem occurs e.g. if there is more than one attributive adjective in a noun phrase. In such cases, it would be helpful to only process a default ordering, knowing that the resulting semantic representations would be logically equivalent. The constraint types defined so far do not provide any possibility to define such a default order. I chose to renounce on introducing a third type of constraint for such cases because both a global default behaviour (such as to always process identically labeled dependents from left to right) and label-specific constraints (such as to always process the second of two *atr1* dependents first) led to conflicts with the Esperanto dependency grammar I worked with, mainly due to the fact that both quantifiers and adjectives recieve the dependency label *atr1* in this grammar.

4.2 Dealing with Arising Complexity Issues

Even with this constraint system, the combinatorical explosion of possible application variants to be considered at each node makes semantic processing rather slow.

Some of these difficulties are due to the fact that in the prototype implementation, I chose to use a very simple approach to enforcing those constraints in order to achieve maximum transparency. For the enforcement of the precedence constraints, the basic idea is to compute all possible permutations of the dependents, check each of them as a candidate order of application and discard each permutation that violates one of the constraints.

There are certainly much more advanced methods to generate all application variants that fulfill a given set of constraints without first generating all the variants and considering them one by one, especially since the constraints imposed here are of a rather predictable nature. But doing so would have required the use of much more advanced constraint programming techniques, which would have ruined the conceptual simplicity of my approach and made my code much harder to understand. For an implementation of the constraint system that is to be of more use than a mere proof of concept, this would certainly be an area worth of investigation.

Another way to improve the performance of the system by reducing complexity is grammar optimization. As a rule of thumb, imposing more constraints makes the system faster, but less robust. For maximum robustness, one would simply not define any constraints at all. Then the system would just apply everything to everything in every order in a brute-force manner, with the disastrous complexity results discussed in the beginning of the preceding section. Robustness would be at a maximum because if there was any way to combine the semantics of the words into a sentential form, the system would find it. On the other hand, if we use constraints to prescribe directionality and order of application for any possible configuration of head and dependents, the computation will be almost as fast as with explicitly defined rules for each configuration. However, an unexpected configuration would almost certainly fail to result in a valid computation even if such a computation were possible in principle, with negative consequences for the robustness of semantic processing.

This trade-off between robustness and speed is so predictable that it could even be used to dynamically adapt grammars for different purposes. One could for example rank the constraints in the grammar from most reliable to least reliable, and adapt the number of constraints taken into consideration according to the momentary needs in terms of speed and robustness. If a message is unlikely to contain unexpected structures and has to be processed fast, one could use as many constraints as possible to speed up semantic processing, whereas if there is plenty of time to process a sentence that is likely to contain rare constructions, one could increase robustness at the cost of runtime by only considering a few very reliable constraints.

5 Robustness and Partial Interpretation

One of the main reasons for using dependency grammar as a syntactic model for natural language processing has always been its robustness on unexpected input. While in a phrase structure grammar, input containing spurious syntactic phenomena is likely not to be recognized, a dependency parser will usually still find a syntactic analysis. Carrying over this robustness at least partially to semantic analysis is therefore a logical next step to attempt once the basics of applying lambda calculus to dependency structures are established.

LARA features two ways of achieving more robust semantic processing. Firstly, the underspecification of the directionality and precedence of functional applications made possible by the constraint system can be used to achieve a better coverage of unexpected or unusual input. Secondly, as will be shown, the system can to a certain extent mitigate the effect of interpretative gaps caused by missing lexical entries. Both of these points will be discussed in detail in this chapter.

5.1 The DPURC-SEM Model of Robust Partial Interpretation

5.1.1 Adding Robustness by Underspecifying Application Rules

When defining semantic rules to determine how the semantics of constituents must combine, for reasons of computational efficiency it is desirable to confine the range of possible applications as much as possible. Defining rules for each constituent configuration represents the extreme approach in this vein. One could think that the constraint system is merely a way of stating many such rules more efficiently. However, there are also cases in which fixed orders of application would exclude some valid interpretations that a constraint system can compute.

The treatment of the negation in the following sentence is such a case:

(3)

Ĉiu	*esploranto*	*ne*	*elpensas*	*novaĵon.*
every-NOM	researcher-NOM	not	invent-PRES	novelty-ACC

'Every researcher does not invent something new.'

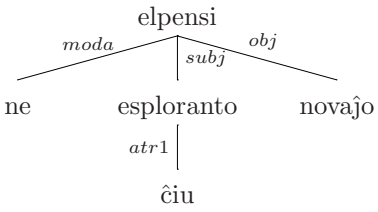

Figure 7: dependency structure for *ĉiu esploranto ne elpensas novaĵon*

The problem here is that there are two possible scopes for the negation:

It can range over the whole sentence, giving rise to the reading

$$\neg\forall x(esploranto(x) \rightarrow \exists y(novaĵo(y) \wedge elpensi(x,y))).$$

Alternatively, the negation has only local scope over the verb, resulting in the reading

$$\forall x(esploranto(x) \rightarrow \neg\exists y(novaĵo(y) \wedge elpensi(x,y))).$$

Since in dependency grammar, the verb is the head of the sentence, it should be possible to compute both meanings from the dependency structure in Figure 7, with the negation as dependent of the verb.

In the semantic lexicon, there are two entries for negation: $\lambda p\neg p$ and $\lambda p\lambda x\neg p(x)$. For sentence scope, we simply need the first term to be applied last after computing the meaning of the sentence without the negation. This means that the verb must be combined with subject and object before being combined with the adverb. For local scope, we need the second term, and this one must be applied to the verb before the object is applied. The two different readings are thus computed by combining the dependents' semantics with the head's semantics in different orders. With a predefined order of application for this configuration of dependents, one of the two readings would have been lost. The underspecification of precedence made possible by the constraint system thus helps to improve semantic coverage.

Another advantage of underspecified application rules is that missing semantic rules do not necessarily cause semantic derivation to fail because there is always the fallback case of simply trying both directions. Consider e.g. the sentence

(4) *Mi* *ne* *rekonas* *lin* *hodiaŭ.*
I-NOM not recognize-PRES him-ACC today
'I do not recognize him today.'

Figure 8 shows the two dependency structures that DPURC finds for this sentence. The problem here is that our dependency grammar allows only one modal adjunct per verb. In the second structure, this place is taken by *hodiaŭ*, such that *ne* can only be interpreted as free adjunct to the personal pronoun *mi*. So the only interpretation we can expect to get is the one meaning "It is not me who recognizes him today". One might criticize this as a flaw in the grammar, but even under these circumstances, the sentence will be interpreted correctly although we have no semantic application rule determining how *ne* is to be interpreted as free adjunct of a pronoun. The system will just try out both versions and check whether one of the two works out. The variant that applies *ne* to *mi* results in the logical form *¬rekoni(mi,li)* for the entire sentence, while the variant that applies *mi* to *ne* results in the syntactically invalid nonsentential form *¬mi(λkλy k(λx rekoni(y,x)))*. By sieving out the nonsentential form, we get *¬rekoni(mi,li)* as logical form for the sentence.

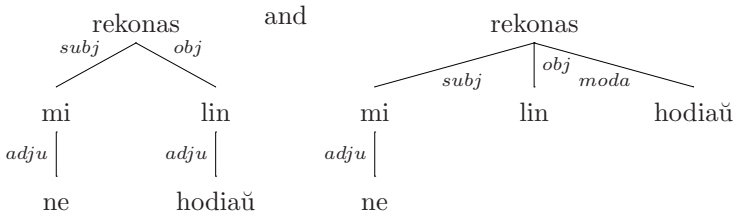

Figure 8: dependency structures for *mi ne rekonas lin hodiaŭ*

This is certainly a correct interpretation, although it may be a little too weak because the implicature that someone other than the speaker recognized the person designated by *li* gets lost during the process. The fact that the information contained in the second adverb is also lost is an instance of partial interpretation (see next section).

As we have seen, a constraint system with the default behavior of trying out all alternatives not forbidden by the constraints proves to be beneficial for the robustness of semantic interpretation.

5.1.2 Dealing with Interpretative Gaps

Quite often, the syntactic role of a word is recognized by the dependency parser without problems, but the word's contribution to the semantics of the sentence remains obscure or is just missing in the semantic lexicon. In some of these cases, it is still possible to derive a valid interpretation of the sentence without knowing the semantic value of the unknown word. Sentences can thus have interpretative gaps and still be usefully interpreted.

An example where this is the case would be the definite article, notorious for being difficult to interpret because of the deictic component in its meaning as well as its dependence on scope and context. BB05 choose not to include the definite article in their fragment of English for this reason. However, the system's refusal to interpret a sentence only because it contains a definite article might be unnecessarily strict. At least in some cases, a part of the meaning can still be extracted from such a sentence. Consider the sentence *The green house is next to the red house.* We certainly cannot know which two houses the speaker is referring to without knowing the context of the utterance, which would require keeping track of discourse contexts. However, instead of just ignoring this sentence because we cannot fully interpret it, we can at least infer that there must be two houses, one red and the other green, that are next to each other. In first-order logic, this could be represented as $\exists v \exists w$ *house(v)* $\wedge$ *house(w)* $\wedge$ *red(v)* $\wedge$ *green(w)* $\wedge$ *next_to(v,w)* .

Even though this is not a complete interpretation, it is certainly better than nothing because it still encodes most of the information contained in the sentence. If we use a model builder to represent the situation described by the speaker, the model builder's tendency to build the minimal model of the description will even ensure that another mention of *the red house* will be interpreted to refer to the same entity in the model, which was originally the speaker's purpose when using the definite article. In Esperanto, this desired interpretation is even identical to the interpretation of the equivalent sentence without the article:
$[\![$*la domo ruĝa estas flanke de la domo verda*$]\!] \simeq$
$[\![$*domo ruĝa estas flanke de domo verda* $]\!] =$
$\exists v \exists w$ *domo(v)* $\wedge$ *domo(w)* $\wedge$ *ruĝa(v)* $\wedge$ *verda(w)* $\wedge$ *flanke_de(v,w).*
This is due to the fact that Esperanto does not have an indefinite article, such that nouns without quantifiers or articles are interpreted to be existentially quantifed over.

Another example are adverbs that are notoriously difficult to predict in their semantic content. If an adverb is not listed in the semantic lexicon, it is usually impossible to determine its contribution to the meaning of a sentence. Often enough, an adverb contributes not much more than the attitude of the speaker to an event. Such adverbs in Esperanto are e.g. *bedaŭrinde* "regrettably", *fakte* "really", and *neatendite* "unexpectedly". Since these adverbs do not contribute anything to the meaning that would be useful for extracting the factual content of utterances, it could be useful to ignore uninterpretable adverbs in sentences we want the computer to understand.

As a third example I would like to mention the treatment of numerals. The semantics of numerals is conceptually rather simple, but even expressing a small number like "five" requires a somewhat large first-order formula:
$\lambda P.\lambda Q.\ \exists v \exists w \exists x \exists y \exists z\ P(v) \wedge Q(v) \wedge P(w) \wedge Q(w) \wedge P(x) \wedge Q(x) \wedge P(y) \wedge Q(y) \wedge P(z) \wedge Q(z) \wedge v \neq w \wedge v \neq x \wedge v \neq y \wedge v \neq z \wedge w \neq x \wedge w \neq y \wedge w \neq z \wedge x \neq y \wedge x \neq z \wedge y \neq z.$

LARA's semantic lexicon does not have entries for the meaning of numerals. However, it is still possible to retrieve some useful information from sentences with numerals.

Consider the following sentence for which DPURC recognizes the dependency structure in Figure 9:

(5) *La hundo prenas nur kvin ostojn.*
the dog-NOM take-PRES only five bone-PL-ACC
'The dog takes only five bones.'

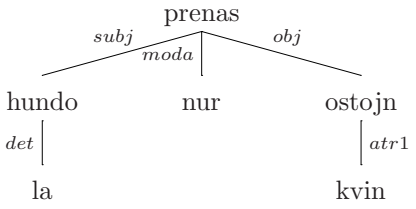

Figure 9: full dependency structure for *la hundo prenas nur kvin ostojn*

In this sentence, we see an example of each of the three cases discussed in this section: the semantic lexicon does not contain entries for the article *la*, the adverb *nur*, or the numeral *kvin*. All of these tokens will simply not be taken into consideration for the derivation of the semantics. Furthermore, our grammar fragment does not contain any plural semantics, so plural forms will simply be interpreted like the corresponding singular forms. The LF for this dependency structure will thus be identical to the LF for the much simpler structure in Figure 10. Because of the implicit existential quantification, this will evaluate to the logical form $\exists x(hundo(x) \wedge \exists y\ (osto(y) \wedge preni(x,y)))$. We have indeed lost some information from the original sentence, but the knowledge that there is a dog who takes a bone is certainly a valid - though extremely weak - interpretation.

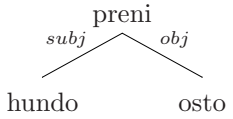

Figure 10: interpretable dependency structure for *la hundo prenas nur kvin ostojn*

By using the defensive interpretation strategy instantiated in the three examples, many complicated sentences that would be really hard to completely interpret at least yield some truth conditions that could be relevant for interpreting a longer text passage. In case it later becomes relevant whether the dog is in possession of any bone, the truth conditions we defensively extracted from the complicated sentence will be useful.

For Esperanto, all of the mentioned examples amount to simply ignoring certain leaf nodes in the dependency structure while computing the semantics. Such a behaviour turned out to be very easy to implement and integrate into the constraint system. I have done so in my prototype implementation by ignoring a node and its dependents if the node's token is not covered by the semantic lexicon.

5.2 Limitations of the approach

As we have seen, simply ignoring uninterpretable tokens during semantic derivation turned out to be surprisingly useful for robust processing. Unfortunately, it is very easy to think of cases where this is not so.

For instance, certainly not all adverbs are semantically void. To a certain degree, those that are not could be listed in the semantic lexicon, but the sheer number of different adverbs in natural languages makes this approach rather tedious. To see the dangers of partial interpretation, consider that the adverb in modal adjunct position might well be something like "allegedly" or "hopefully", both of which have a severe impact on the information content of sentences. Ignoring such adverbs makes formal interpretations useless, if not misleading.

One might argue that for such cases, it might be better not to interpret the sentence at all rather than to risk an interpretation that is so completely wrong. It is difficult to tell whether the danger of lax interpretation outweighs the benefits of robust interpretation here.

The problem gets even worse for e.g. prepositions. Some of them are always difficult to interpret because they have different temporal, local and abstract meanings. For a system without world knowledge, it is therefore often impossible to tell how to interpret a prepositional phrase. The partial interpretation approach to this would be to simply ignore prepositions as well as their dependents. For sentences like "the dog sleeps in its kennel", this is certainly not a problem because we still get some correct information out of the sentence. But for more abstract cases such as "he is by no means a weak opponent" or "I can see him in my dreams", ignoring the semantics of the prepositional phrases has severe impacts on the correctness of interpretation, up to the extreme case that the computed semantics is even the negation of the intended meaning.

Such problems might severely limit the usefulness of partial interpretation, but it might also be the case that the most extreme cases can be covered by explicit semantic rules and that the majority of cases is harmless. It would certainly be worthwile to perform quantitative studies on corpus data to assess the potential usefulness of partial interpretation.

5.3 Other Approaches to Robust Deep Semantics

Lev et al. (2004) present initial work on a system that is designed to solve logic puzzles in natural language. As they demonstrate, logic puzzles are an ideal test case for deep semantics because solving a logic puzzle usually requires quite a bit of reasoning in first-order logic. However, interpretation must be as complete as possible, any information the computer cannot extract from the puzzle description might lead to multiple solutions. This is why wide coverage and robust processing are essential for success. Logic puzzles contain a large variety of syntactic constructions that are hard to cover using hand-built rules, which is why they rely on statistical syntactic parsing. To achieve robust deep semantics, they start out with the BB05 semantic processing system (which they reimplement in Lisp) and take a similar approach to designing a generic semantic lexicon.

Lev et al. (2004) address the problem of syntactic structures for which no semantic templates exist by trying out all combinatorical alternatives by default. Since they operate on largely binary phrase structure trees and use a typed version of lambda calculus, the combinatorics of this approach are a lot less bad than in my approach on dependency structures, which makes it unnecessary for them to devise a constraint system to steer the process. Special treatment for VPs and NPs further reduces the complexity of the problem: VPs in general only export an event variable instead of variables for subject and object, NPs are always analyzed as generalized quantifiers.

5.4 Other Attempts at Partial Interpretation

Coheur et al. (2004) present a syntax/semantics interface for dependency grammar that addresses some difficulties of semantic computation on dependency structures similar to the problems discussed here. They are also concerned with the problems arising from a too strong connection between syntactic and semantic rules, and also solve the problem by stating rules for semantic computation independently. However, their formalism is less concerned with preserving locality of computation: in order to select semantic rules for application, their system AsdeCopas relies on wider contexts. Partial interpretation is achieved by a subsumption hierarchy of semantic rules in which more specific rules (i.e. rules selected by more specific contexts) have priority over less specific ones. The least specific default rules that are applied on unexpected structures do not take into account the whole dependency structure, but the interpretable parts. This approach to partial interpretation certainly has the advantage of being easier to control than the DPURC-SEM approach, but it also means that uninterpretable structures that are to recieve partial interpretations must to a certain extent still be expected and endorsed by the rule system. The less ambitious and more careful approach to wide-coverage interpretation taken by AsdeCopas will probably yield better results for a small range of structures, but might fail on quite a lot of unexpected input that could be interpreted by a less restrained system like DPURC-SEM.

Milward (1992) introduces the formalism of DDG (Dynamic Dependency Grammar) in which the incremental construction of dependency structures is expressed in the form of transitions between states. The states contain feature structures defining syntactic types with information about the structure already seen as well as about expected structure that would be necessary to yield a complete sentence. Although his main concern is syntactic parsing, Milward also proposes to augment each state with a semantic type and a semantic value. In his view, a rather direct mapping between syntactic and semantic types could be established. Given these types, lambda terms for each state of syntactic parsing could relatively easily be generated. This approach to generating lambda terms from dependency structures is only demonstrated in one trivial example, and it does not become clear how this kind of approach could be applied to a larger set of structures. However, the approach gives rise to important ideas for partial interpretation. Since the syntactic states contain information on missing structure, they could help to yield default semantics for uninterpretable substructures. In this way, it would be possible to achieve more robust partial processing than with DPURC-SEM. It would not be necessary to confine partial interpretation to ignoring uninterpretable parts. Instead, one could provide generic interpretations e.g. for missing objects or preposition arguments. Unfortunately, DPURC-SEM cannot easily be enhanced by such capabilities because without any lexical information e.g. about the valency of verbs, it is virtually impossible to determine missing parts in dependency structures.

LARA	CURT	effect
`interpretoj`	`readings`	prints current readings
`elektu N`	`select N`	select a reading (N should be a number)
`denove`	`new`	starts a new discourse
`historio`	`history`	shows history of discourse
`modeloj`	`models`	prints current models
`resumo`	`summary`	eliminate equivalent readings
`scio`	`knowledge`	calculate and show background knowledge
`infikso`	`infix`	display formulas in infix notation
`prefikso`	`prefix`	display formulas in prefix notation
`helpo`	`help`	display list of commands
`gxis`	`bye`	no more talking

Figure 11: LARA commands and their CURT equivalents

6 Implementation

This chapter describes LARA as a prototype implementation of the ideas discussed in the previous chapters. The implementation builds heavily on the CURT system and has been developed in SWI Prolog. External components necessary to run and test the system are the theorem prover `otter`, the model builder `mace2`, and the dependency parser DPURC. In case of difficulties, it is advisable to first follow the BB05 instructions to install CURT, the migration to LARA should then be not much of an issue.

After a quick introduction to LARA in the form of a sample dialogue, I systematically describe all the parts that have changed in comparison with CURT. Starting with a quick overview of the syntactic parsing component, I continue with a discussion of the dependency grammar for Esperanto that provides the system with syntactic structures. The section thereafter introduces the semantic lexicon and the semantic constraints for Esperanto that the DPURC-SEM semantic processing engine uses to compute interpretations for these structures. Section 6.5 then describes the implementation of DPURC-SEM in detail, and the last section collects a few ideas on how the LARA system could be extended to be a better testing environment for robust deep semantics.

6.1 Introducing LARA

The demo application for the DPURC-SEM engine implemented for testing purposes is an esperantized version of CURT carrying the name LARA. The only version of LARA that comes packaged with the code for this thesis is `helpemaLara`, which is roughly equivalent to `helpfulCurt` without the world knowledge component, but with much wider lexical coverage.

After loading `helpemaLara.pl`, the user is prompted to start a dialogue with LARA by entering `lara.` Having done this, the user has the same options as in `helpfulCurt`, the only difference being that the command names are now in Esperanto (for a quick reference, see Figure 11).

To illustrate LARA's modest abilities, here is the protocol of a small dialogue with the system: [1]

```
> mi amas cxiun hundon kiu amas min

Lara: Okej.
```

[1] For compatibility reasons, LARA uses the so-called x-convention (*iksokodo*) to express Esperanto diacritics; this also explains why e.g. *ĉiu* must be written `cxiu` to communicate with LARA.

I start to describe a situation by stating that "I like every dog who likes me". After a few seconds (LARA has a lot to compute for such a relative clause), the system reports back stating that it understood the message. Note that CURT is not able to understand the equivalent relative construction in English, but the constraint system manages to extract the correct meaning from the dependency structure. I continue by introducing one specific dog named Fido into the discourse. This is achieved by saying "Fido is a dog":

```
> fido estas hundo

Lara: Okej.

> interpretoj

1 and(  all(A, imp(and(ami(A, mi), hundo(A)), ami(mi, A))),
        some(B, and(fido(B), some(C, and(hundo(C), eq(B, C))))) )
2 and(  all(A, imp(and(ami(A, mi), hundo(A)), ami(mi, A))),
        some(B, and(fido(B), eq(B, hundo))) )
3 and(  all(A, imp(and(ami(A, mi), hundo(A)), ami(mi, A))),
        some(B, and(hundo(B), eq(fido, B))) )
4 and(  all(A, imp(and(ami(A, mi), hundo(A)), ami(mi, A))),
        some(B, and(hundo(B), some(C, and(fido(C), eq(B, C))))) )
5 and(  all(A, imp(and(ami(A, mi), hundo(A)), ami(mi, A))),
        some(B, and(fido(B), eq(hundo, B))) )
6 and(  all(A, imp(and(ami(A, mi), hundo(A)), ami(mi, A))),
        some(B, and(hundo(B), eq(B, fido))) )
7 and(all(A, imp(and(ami(A, mi), hundo(A)), ami(mi, A))), eq(hundo, fido))
8 and(all(A, imp(and(ami(A, mi), hundo(A)), ami(mi, A))), eq(fido, hundo))

> elektu 6
```

Because the parser has no way of recognizing *fido* as a proper name, for the system there are ambiguities e.g. between the readings "there is something that is a *fido* as well as a *hundo*" and "the entity called *fido* is a *hundo*". This results in quite a few possible readings, I am forced to disambiguate manually in order to tell LARA what I mean. The next thing I want to tell the system is that "Fido likes me":

```
> fido amas min

Lara: Okej.

> interpretoj

1 and(  and(all(A, imp(and(ami(A, mi), hundo(A)), ami(mi, A))),
        some(B, and(hundo(B), eq(B, fido)))), some(C, and(fido(C), ami(C, mi))))
2 and(  and(all(A, imp(and(ami(A, mi), hundo(A)), ami(mi, A))),
        some(B, and(hundo(B), eq(B, fido)))), ami(fido, mi))

> elektu 2
```

Again, the system does not know whether I am using *fido* as a proper noun or as a common noun. This time, I only have to decide between two readings, where the second reading that treats *fido* as a constant is the desired one.

It is now possible to demonstrate LARA's capability of detecting inconsistent Esperanto input. The most obvious way to make my story inconsistent is to state that "I do not like Fido":

```
> mi ne amas fidon
```

```
Lara: Ne! Mi ne kredas tion!
```

As expected, LARA detects the inconsistency and refuses to believe me. As we have seen, LARA has about the same capabilities as CURT, with additional strengths and weaknesses resulting from the generalized lexicon made possible by the nice properties of Esperanto morphology.

6.2 The DPURC Dependency Parser

For the biggest part of this thesis, the syntactic parsing component DPURC has been treated as a black box. There are good reasons to do so since the internal workings of the DPURC dependency parser are documented in Dellert (forthcoming). Here, I will only discuss some properties of DPURC that had an impact on major design decisions for the DPURC-SEM system.

Most importantly, the DPURC grammar format influenced the design of the Esperanto dependency grammar. A DPURC grammar consists of rules, each defining one possible link of a dependent to a head (see examples in the next section). It is possible to express uniqueness and ordering restrictions in this format, but non-local context cannot be used to impose restrictions on the applicability of the rules. To use DPURC, it was necessary to express the Esperanto grammar by Schubert (1989) in this format. Since his grammar uses a similar concept of rules, this was rather straightforward for most parts. However, some non-local constraints restricting overgeneration in that grammar were impossible to express in the DPURC formalism, resulting in hundreds of dependency structures even for very short sentences, most of which were completely uninterpretable. Since the emphasis of the prototype system lies on semantic processing and not on syntactic parsing, only a core set of syntactic rules has been taken over into the implementation to alleviate this problem.

When work on DPURC-SEM began, DPURC was already a fully functional and complete component that could be integrated without any internal changes. For that reason, despite the similarity in names, DPURC is in fact an external component. Unlike the DCG component of CURT, DPURC does not build representations during syntactic parsing already. Its only task is to find dependency structures that are subsequently used by DPURC-SEM as guidance for semantic derivation. This means that DPURC-SEM sets in at complete dependency structures, no semantic computation takes place before a syntactic parse is complete. There are two main reasons for this design decision. Firstly, DPURC must backtrack quite a lot to find valid dependency structures, and performing semantic computations on each of the search tree's many branches and twigs would waste a lot of computing time. Secondly, separating the task of syntactic parsing entirely from the task of semantic processing keeps down the complexity of both tasks and allows the system to stay modular.

6.3 Lexicon and Syntax for Esperanto

The lexicon, the morphology and the dependency syntax for Esperanto are stored in DPURC grammar format in `dpurc-gr-eo.pl`. The complete contents of this file can be found in the appendix, for a general description of the grammar format with a broader perspective on the ideas behind it see Dellert (forthcoming).

The DPURC grammar file consists of two major sections: the first one encodes the lexical entries necessary for category lookup of individual tokens, the second section represents a list of possible dependencies, each one defined by a head category, a dependent category and a number of side conditions.

The lexicon is basically a huge collection of Prolog clauses defining the predicate `word/2`. Closed word classes are defined and their members enumerated first, the clauses for open word classes follow afterwards. Some of these clauses have the form of rules that reduce inflected forms to their

canonical forms. In the case of Esperanto, this means reducing all the inflectional forms of nouns and adjectives to the nominative singular and all the inflectional forms of verbs to the infinitive.

A typical example entry from a closed word class is the following:

```
word(trans,  [prp,trans]) :- !.
```

This entry contains the information that the token *trans* is a form of the lemma `trans`, which is of category `prp`, i.e. *preposition*. All members of the closed word classes must be listed because unlike for the open word classes, Esperanto does not provide clear markers there. The cut is necessary to prevent alternative analyses by morphological rules.

For the open word classes, the presence of unique morphological markers for each class makes it feasible not to provide huge word lists but to rely solely on morphological analysis instead. The extreme regularity of affixation without any phenomena like assimilation or vowel harmony that complicate morphological analysis in other languages reduces this task to cutting off suffixes from strings, such that the whole inflectional morphology of Esperanto can be described in a mere 15 Prolog clauses. I will discuss two of these clauses here, the complete list can be found in the appendix.

```
word(Noun, [sub,Lemma,sg,acc]) :-
    sub_atom(Noun,Index,_,0,on),
    CutIndex is Index + 1 ,
    sub_atom(Noun,0,CutIndex,_,Lemma).

word(Verb, [vrb,Lemma,fut]) :-
    sub_atom(Verb,Index,_,0,os),
    sub_atom(Verb,0,Index,_,Stem),
    atom_concat(Stem,i,Lemma).
```

The first clause contains the information that tokens ending in *-on* are to be considered accusative singular forms of nouns. In the first condition, I use the `sub_atom/5`[2] predicate to look for an occurrence of "on" in the word and store its position in the variable `Index`. Since the "o" is actually part of the word's canonical form (the nominative singular for nouns), the lemma must include the "o" from the ending. Thus, the `CutIndex` is defined to be by one larger than `Index`. Using `CutIndex`, we can use another call to `sub_atom/5` to isolate the lemma. The analysis recieves the features `sg` and `acc` to express that the token has been analyzed as the accusative singular form of the lemma.

The second clause defines that tokens ending in *-os* are to be analysed as future tense forms of verbs. Again, the position of the suffix is stored in the variable `Index`. This time, the canonical form (the infinitive) must be formed by replacing the *-os* ending with *-i*. For that purpose, the word is cut at position `Index` to yield the `Stem` to which I append *-i* by means of `atom_concat/3` in order to produce the lemma. The feature `fut` expresses that the system analyses the token as the future tense form of the lemma.

All the morphological rules in the lexicon follow one of these two schemata.

The legal syntactic structures for the dependency grammar are encoded as clauses of the `dh/6` predicate. Each clause defines a legal dependency link with some side conditions. The general format for such rules is

```
dh(Dependent,Head,Headedness,Projectivity,Uniqueness,Label).
```

[2] `sub_atom(+Atom, ?Before, ?Len, ?After, ?Sub)` is an ISO predicate for string processing; the semantics is that `Sub` is the subatom of `Atom` that starts at position `Before`, comprises `Len` characters and stops before the last `After` characters of `Atom`; for more details on usage see e.g. Wielemaker (2008), section 4.21.

In this format, `Dependent` and `Head` impose restrictions on the head and the dependent of the dependency link, respectively. Both variables have to be instantiated with a word description in the format of the second argument to `word/2`, i.e. a list headed by `[Category,Lemma]`, followed by a number of features that vary between word classes.

`Headedness` must have one of the three values `free_order`, `head_initial`, and `head_final`. With these values, it is possible to impose ordering restrictions on the two words linked by the dependency. For instance, `head_initial` forces the head to occur before the dependent, otherwise the link will not be established. Since Esperanto is a language with rather free word order, in most cases any order is allowed. One counterexample is the article that must come before the noun.

`Projectivity` must have either the value `yes` or `no`. Setting this to `no` allows the dependent to be head of a discontinuous constituent, while `yes` forbids this. This can be useful in languages such as Latin, where a verb can occur between its object and the attribute to the object:

(6) *Serpentem* *inveni* *periculosum.*
snake-ACC-SG-M find-1SG-PERF dangerous-ACC-SG-M
'I have found a dangerous snake.'

Discontinuous constituents are very expensive to parse, the general problem for dependency grammars has even been shown to be NP-complete (see Neuhaus and Bröker (1997) for details). Since such constructions are not very common in Esperanto and constantly looking for them slows down syntactic parsing considerably, all the rules contained in the dependency syntax used by LARA define their dependents to be projective.

`Uniqueness` must also have either the value `yes` or `no`. This side condition specifies whether a single head can be linked to more than one dependent using a link with the same label. A typical case where one would want this restriction to apply is the case of subject and verb, where a verb must not have more than one subject. On the other hand, a noun can easily have more than one attribute, e.g. a possessive pronoun and an adjective.

`Label` contains the rule-specific label that a dependency link recieves if it can be established by a rule. Several rules may share one label, but the uniqueness constraints take all instances of dependency links with the same label into account. `Label` is actually not a condition to check, but a piece of information encoded in the syntactic rules that is present in the constructed dependency structure and is later used by the semantics component to decide on the functional applications between lambda terms.

Here are two examples of dependency rules from the Esperanto dependency grammar:

```
% Pronoun as object of a verb
% --> Li aĉetis ĝin.
dh([prn,_,_,acc],[vrb,_,_],free_order,yes,yes,obj).

% Article as determiner of a noun
% --> la domo
dh([art,_],[sub,_,_,_],head_final,yes,yes,det).
```

The first rule specifies that a pronoun in accusative case is allowed to be linked as `obj` dependent to a verb if the constituent headed by the pronoun is continuous and the verb is not yet linked to another `obj` dependent. The object pronoun is allowed to stand both before and after the noun.

The second rule specifies that an article is allowed to be linked to a noun as `det` dependent if it comes before the noun. In Esperanto, case and number of the noun are not mirrored in the form of the article, which explains the anonymous variables in the feature structure of the noun.

The fragment of Esperanto syntax currently covered by LARA is encoded in 30 such rules.

6.4 Semantic Entries and Constraints for Esperanto

In the LARA architecture, both the semantic lexicon and the constraints on semantic processing are defined in the file `eo-dict-sem.pl`. The semantic lexicon works almost exactly as in BB05, and I will only explain some rules where I had to make changes because of the differences between the semantics of English and Esperanto or because of the different approach to the syntactic lexicon.

As explained in previous chapters, the semantic rule system from BB05 has been entirely replaced by a constraint system. The discussion of the rules for that system forms the second part of this section, whereas its internal workings are discussed in detail in the next section.

The semantic lexicon is encoded in clauses that define the `semLex/3` predicate. In principle, these clauses define how to construct or retrieve the lambda expressions for a given lemma and a given category. Some of these clauses are very simple and more or less directly mirror their equivalents that BB05 defined for English:

```
% Prepositions

semLex(prp, Lemma, Sem) :-
   Sem = lam(K,lam(P,lam(Y,and(app(K,lam(X,F)),app(P,Y))))),
   compose(F,Lemma,[Y,X]), !.
```

If this clause is called e.g. with the instantiation `semLex(prp,super,Sem)`, `Sem` will recieve as value the lambda expression $\lambda k \lambda p \lambda y\ k(\lambda x\ super(y, x)) \wedge p(y)$ in the Prolog format that the BB05 implementation of the untyped lambda calculus uses internally. For more information on how this works, see (Blackburn and Bos, 2005, p. 92).

Some of the entries in the semantic lexicon could not so easily be adapted from BB05. The main cause for this are systematic ambiguities in the interpretation of nouns and verbs that are due to the approach taken to the lexicon. As we have seen, large parts of the lexicon are not stated explicitly, but encoded in morphological rules instead. Unfortunately, in Esperanto there are no morphological differences between proper nouns and common nouns or between transitive and intransitive verbs. As a result, those categories cannot be distinguished by the dependency parser. One could certainly devise intelligent methods to alleviate this problem at least on the semantic side by inspecting whole substructures to decide which semantic representations to use, but for this prototype implementation a much simpler approach turned out to be sufficient. The solution adopted here is to simply try both versions for each verb and then see which of the two choices results in sentential forms. The following code shows the rule for verbs:

```
% complex  lexical rule for verbs that can either be transitive or intransitive

semLex(vrb,Lemma,Sem):-
  % Intransitive Verb
  (
    Sem = lam(X,Formula),
    compose(Formula,Lemma,[X])
  );
  % Transitive Verb
  (
    Sem = lam(K,lam(Y,app(K,lam(X,Formula)))),
    compose(Formula,Lemma,[Y,X])
  ), !.
```

The disjunction of the two cases has the effect that the two possible interpretations are processed in any case. Prolog's backtracking mechanism will ensure that all the rest of the semantic derivation is computed with both alternatives. An analogous approach applies to the problem of proper nouns vs. common nouns (see appendix for details). The major disadvantage of this approach is that it

either creates a lot of interpretations that are nonsentential (in the verb case) or it creates multiple sentential forms that the user must then disambiguate by hand (as in the noun case). Especially the last problem is difficult to address because there are no comprehensive lists of proper nouns in Esperanto and because proper and common nouns do not differ enough in syntactic behaviour as opposed to e.g. in English where only proper nouns can occur without a determiner in singular.

All the entries in the semantic lexicon discussed so far end with a cut. The last lexical entry provides an explanation for this:

```
semLex(_,_,unknown).
```

This rule defines a default representation for uninterpretable tokens. The derivation system (see next section) knows that it cannot calculate with such a representation and simply ignores it during derivation, which is the mechanism my implementation uses to deal with interpretative gaps. However, since the two anonymous variables in the clause match any category and any lemma, even tokens who were already covered by the semantic lexicon would recieve `unknown` as a second representation. This would result in a lot of spurious incomplete interpretations (namely the interpretations of all structure fragments with the same head) where the original intention simply was to interpret as much as possible. To ensure that the default interpretation only applies to tokens that couldn't be interpreted otherwise, it is necessary to put a cut after each entry.

The application constraints are defined in the second part of the file. Each of the two types of constraints discussed in chapter 4 is encoded in a predicate: `directional/4` for directional constraints and `precedence/3` for precedence constraints. Some instances of each will be discussed in the remainder of this section.

`directional/4` clauses have the following general form:

```
directional(Head:HeadSem, Label, Dep:DepSem, Instruction).
```

A typical simple directional constraint is the following:

```
directional(vrb:A,  obj,  sub:B,  app(A,B)).
```

This basically states that a head of category `vrb` with the semantic representation `A` that has a dependency link of type `obj` to a word with category `sub` and semantic representation `B` is processed semantically by applying `A` to `B`. This is completely analogous to the rule for English VPs of the form (VP (V) (NP)), where the semantics of the verb is also applied to the semantics of the object NP to yield the semantics of the entire VP. However, we are still dealing with dependency structures here, which means that we apply semantics along dependency links, not between sister nodes. Moreover, there could well be other dependents to the verb, e.g. a subject or a modal modifier that would also have to be applied to the verb or vice versa. To determine the order of application for multiple dependents is the task of the precedence constraints.

One might ask why the instruction in the fourth argument of `directional/4` is stated as a lambda term and not just as a binary flag deciding on the direction of application. This is the result of a trade-off between formal clarity and elegance of implementation when trying to cover implicit existential quantification. As already mentioned, common nouns in Esperanto can constitute complete noun phrases even without any determiner. In such cases, an implicit existential quantification must occur. Two different ways to simulate this behaviour were easy to integrate with the system, only one of which worked out. The first attempt was to give a third possible interpretation to each noun, namely a lambda term of the form $\lambda q \exists x(q(x) \wedge dog(x))$, which worked out for simple test cases. However, this approach immediately reached its limits when an attributive adjective was added to the implicitly quantified noun because in the BB05 system, attributes have to be applied to nouns before any quantifiers come in. This only left open the second possibility, which was not to introduce the implicit quantification at the lexical level, but as an alternative processing

instruction for the interpretation of noun phrases as subjects or objects. At this point, the possibility to give application instructions in the form of lambda terms was very useful. The variant of our example constraint with the alternative application instruction for implicit quantification looks like the following:

```
directional(vrb:A,  obj,    sub:B,
 app(A,app(lam(U,lam(V,some(X,and(app(U,X),app(V,X))))),B))) :- !.
```

This should perhaps be called an application rule rather than a constraint because it contains a complicated instruction that could be paraphrased as "first apply the semantics of an existential quantifier to the object NP, then apply the verb semantics to the result". Nevertheless, the computational treatment of the two versions does not differ in the least bit, since both versions simply use lambda terms in BB05 format to define how the semantic representations are to be combined.

To achieve the robustness discussed in section 5.1.1, we need to specify a default behaviour that simply tries application in both directions if no more specific instruction was encountered. The following two clauses have exactly that effect:

```
directional(_:A,    _,      _:B,    app(B,A)).
directional(_:A,    _,      _:B,    app(A,B)).
```

Again, we need cuts in the body of each more specific constraint to enjoy the computational benefits of the constraint system, because otherwise all combinatorically possible functional applications would be tried out even if more specific instructions were found, resulting in the combinatorical explosion discussed in section 4.2.

The other type of constraints that the system uses to guide semantic processing are precedence constraints. These constraints are stated as clauses defining the `precedence/3` predicate according to the following schema:

```
precedence(HeadCategory,   Label1,    Label2)
```

The semantics of such a constraint is that when processing a node with category `HeadCategory`, no dependency link with label `Label2` must be processed before any dependency link with label `Label1`. Consider for example the following precedence constraint:

```
precedence(vrb,   subj,    proa).
```

This constraint states that `subj` dependents (i.e. subjects) must be combined with their verbal governor before `proa` dependents (i.e. propositional adjuncts). This ensures that the interpretation of a sentence is complete before it is combined with the interpretation of a second one. This constraint interacts with another precedence constraint:

```
precedence(vrb,   obj,    subj).
```

This means that when computing the semantics of a verbal head, the processing of `obj` dependents must precede the application of `subj` dependents. To illustrate the interaction of these two precedence constraints, consider the case of a verbal head with three dependents of type `subj`, `proa`, and `obj`. With three dependency links to process, $3! = 6$ different orders of application are possible. The first constraint only allows the orders `[subj,obj,proa]`, `[subj,proa,obj]`, and `[obj,subj,proa]`. The second constraint only allows the orders `[obj,subj,proa]`, `[obj,proa,subj]`, and `[proa,obj,subj]`. The constraint system only allows those orders of application that are allowed by each of the constraints, so in this case the order `[obj,subj,proa]` would be the only one that is tried out. The seven precedence constraints that are currently part of the system are carefully engineered to always allow the correct order of application and to disallow as many incorrect orders as possible. For that reason, the two interacting constraints only allow the correct order of functional applications in this case.

6.5 The DPURC-SEM Engine

This section describes the core of the DPURC-SEM semantic processing system that resides in the file `dpurc-sem.pl`. This is in many ways the central component of the prototype system because the semantic constraints defined in the grammar are applied here to compute logical forms for dependency structures.

The predicate that LARA uses internally to retrieve the possible readings for a list of input tokens is `syn_sem_parse/2` which in turn calls `syn_sem_single_parse/2`:

```
syn_sem_parse(Input, Sems) :-
  setof(Sem,syn_sem_single_parse(Input,Sem),AllSems),
  filterAlphabeticVariants(AllSems,Sems).

syn_sem_single_parse(Input, Sem) :-
  parse(Input,[Head]),
  sem_single_parse(Head,Sem).
```

The first predicate aggregates all the different semantic interpretations that `syn_sem_single_parse/2` produces into the set `AllSems` and then eliminates all the alphabetic variants still in the set using the `filterAlphabeticVariants/2` predicate by BB05, such that the set of interpretations `Sems` only contains syntactically different logical forms. It would also be desirable to eliminate all logically equivalent forms, but there are sometimes so many interpretations that the costs in computing time for such an approach quickly become unaffordable.

The `syn_sem_single_parse/2` predicate uses the DPURC predicate `parse/2` to retrieve one dependency structure whose root is stored in the variable `Head`. This dependency structure is then handed on to the `sem_single_parse/2` predicate, which returns a single interpretation `Sem` that `syn_sem_single_parse/2` passes on. Via the backtracking mechanism, `syn_sem_parse/2` thus aggregates all the different interpretations of all the different dependency structures over the `Input` list.

The next step is to see what `sem_single_parse/2` does to compute the logical forms:

```
sem_single_parse(Head, Converted) :-
  sem_proc(Head, Sem),
  betaConvert(Sem,Converted),
  \+ lambda_or_app_occurrence(Converted) .
```

The main processing step is hidden behind the call to `sem_proc/2`, which takes the `Head` of a dependency structure as input, interprets it and returns a lambda expression `Sem`. This is usually a huge lambda expression that then melts down while being beta-converted by means of the `betaConvert/2` predicate as defined by BB05. In CURT, the result of beta conversion was guaranteed to be sentential because the grammar was intricately balanced to only parse structures that could be completely and safely analysed. Because of the robust processing approach of DPURC-SEM, many of the beta-converted lambda expressions tend to be non-sentential or unsatisfied, i.e. they still contain instances of the `lam` and `app` predicates somewhere. For robust processing, this problem is inevitable, and a straightforward way to deal with this is to sieve out non-sentential logical forms in a postprocessing step. Exactly this is achieved by the negation-by-failure call to the predicate `lambda_or_app_occurrence/1` that fails if its single argument is a sentential logical form.

We have now arrived at `sem_proc/2`, the central predicate for semantic processing:

```
sem_proc([Label,_,_,_,_,Dependents,_,Cat,Lemma|Rest], Sem) :-
  Dependents = [],
  semLex(Cat, Lemma, Sem).
```

```
sem_proc([Label,_,_,_,_,Dependents,_,Cat,Lemma|Rest],Sem) :-
  compute_dep_sem(Dependents, [], DepSems),
  permutation(DepSems, InverseDepSemsVariant),
  \+ violates_precedence(Cat,InverseDepSemsVariant),
  reverseList(InverseDepSemsVariant, DepSemsVariant),
  semLex(Cat,Lemma,HeadSem),
  apply_directionally(Cat, HeadSem, DepSemsVariant, Sem).
```

This predicate serves to compute the semantic interpretation of a dependency structure by recursively descending into it and building up the logical forms in the ways specified by the dependency tree and allowed by the semantic constraints. The first clause is the base case: For a node without dependents, a simple lookup in the semantic lexicon via the `semLex/3` predicate suffices to retrieve the semantic representation for that dependency node. The exact structure of the lists that represent the nodes in the dependency structure is not really important here, for more details please consult the DPURC documentation, i.e. Dellert (forthcoming).

Computing the semantics certainly gets a little more complicated in the recursive case. First of all, the semantics of all the dependents must be computed recursively and stored in the `DepSems` list. This list is then permuted in every possible way by means of a standard `permutation/2` predicate, and each permutation is interpreted as an inverse order of functional applications. The order is inverse because it is much easier to check for violation of precedence constraints if the list is interpreted as an inverse order of applications, with the head of the list containing the lambda term that is to be applied last. Once a permutation has passed the check against violation of precedence constraints, it must be reversed to yield a list of application instructions that can be read from head to tail. Finally, the semantic entry for the head is retrieved and handed on to the predicate `apply_directionally/4` along with the head's category and the order of applications defined by the legal permutation. This predicate makes use of the directionality constraints as a guide through the computation process and combines the semantics of the head with the semantics of the dependents in the order defined by the permutation:

```
apply_directionally(_,HeadSem,[],HeadSem).

apply_directionally(_,unknown,_,unknown).

apply_directionally(HeadCat,HeadSem,[Label:unknown|OtherDepSems],FinalSem) :-
  apply_directionally(HeadCat,HeadSem,OtherDepSems,FinalSem).

apply_directionally(HeadCat,HeadSem,[Label:DepSem|OtherDepSems],FinalSem) :-
  directional(HeadCat:HeadSem,Label,_:DepSem,NextSem),
  apply_directionally(HeadCat,NextSem,OtherDepSems,FinalSem).
```

The first clause is the base case for list traversal: After the semantics of all the dependents has been taken into account, the resulting HeadSem is returned. The second clause defines the behaviour if the HeadSem could not be found in the lexicon: the entire constituent becomes uninterpretable, the head's governor will find an `unknown` dummy semantics in its `DepSems` list. This in turn is the case covered by the third clause: Uninterpretable dependents are simply ignored during the computation. In the last clause, finally, the directional constraints come into play: If there is an interpretable dependent heading the `DepSems` list, the system tries to find instructions on how to combine the semantics of head and dependent by looking at the `directional` constraints. If such an instruction was found (as we have seen, the two simple default instructions will be found otherwise), this instruction is stored in `NextSem` and already represents the new head semantics after the combination of head and dependent. Thus, the recursive call to `apply_directionally/4` must combine the remainder of the `DepSems` list to the new head representation stored in `NextSem`.

One predicate that still needs to be explained is `violates_precedence/2`:

```
violates_precedence(HeadCat,[First:_|DepSemsSequence]) :-
 ( precedence(HeadCat,First,Preceded),
   memberList(DepSemsSequence,Preceded:_));
 violates_precedence(HeadCat,DepSemsSequence).
```

This predicate detects violations of the precedence constraints in inverse sequences of dependent semantics. The basic idea is to check for each item in the list if it must precede something that comes before it in the order (i.e. after it in the inverse list). We start with the first item in the list (i.e. the dependent semantics that would be applied last) and check whether there are any precedence constraints enforcing that this item must precede another item. If there are are such constraints, we check for violations by checking if the remainder of the list (i.e. the dependents that would be applied before the current dependent) contains the other item. If we could not find a constraint violation that involved the first item in the list, we make a recursive call on the remainder of the list to find a violation involving the second item instead. We continue like this until we find a violation or fail because the `DepSemsSequence` has been completely processed by recursive calls, in which case we can be sure that there can be no violation of any precedence constraint in the `DepSemsSequence`.

The last important predicate to explain is the structure traversal predicate `compute_dep_sem/3`:

```
compute_dep_sem([],DepSems,DepSems).

compute_dep_sem([Dep|Dependents], DepSems, ReturnDepSems) :-
  \+ var(Dep),
  sem_proc(Dep, DepSem),
  Dep = [Label|_],
  append(DepSems,[Label:DepSem],NewDepSems),
  compute_dep_sem(Dependents, NewDepSems, ReturnDepSems).
```

This predicate recursively computes the semantic representations for an open list of dependents. It is used by `sem_proc/2` to retrieve the dependent semantics to combine with the head semantics of the current node. Essentially, the predicate defines a linear traversal of an open list. An open list is a list with a variable as its tail. In DPURC, open lists with unified tails are used for structure sharing between parsing components, therefore the dependent lists that by recursion constitute the dependency structures are such open lists. At the end of the list, we hence do not arrive at an empty list, but at a variable. This is why the variable check on the list's head implements the base case for the list traversal recursion. The other base case only covers the case of a true empty list of dependents. The rest of the predicate is straightforward: The semantic representations for all the dependents are computed by calling `sem_proc/2` via indirect recursion and subsequently appended to a result list. `compute_dep_sem/3` is in a sense the driver predicate that steers the traversal of the dependency structure, while `sem_proc/2` does the work of combining the semantics according to the constraints.

This concludes the discussion of the DPURC-SEM system. For testing purposes, the reader might want to load `dpurc-sem.pl` into SWI Prolog and experiment with the two testing predicates `try_sem/1` and `try_all_sem/1`. Both predicates expect an Esperanto sentence in the form of a list of tokens as input. `try_sem/1` will list the dependency structures found by DPURC along with the logical forms DPURC-SEM was able to extract from each of them, while `try_all_sem/1` will just enumerate the readings for that sentence as LARA would see them.

6.6 Outlook: User Feedback for Partial Interpretations

At the moment, when LARA is confronted with a sentence that can be only partially interpreted, the system will just tacitly assume that the partial interpretation of the sentence is correct. For the user, there is no possibility to even see that there was something in the sentence that LARA could not completely understand. It is easy to imagine that after recieving an "`Okej.`" message

as answer to entering "`la hundo prenas nur kvin ostojn`", the user assumes that LARA has completely understood the sentence and now knows that the dog has taken five bones, while LARA really knows only about one. A user could even enter a sentence that is accepted and would only find out much later that LARA has understood the exact opposite of what was meant. The behaviour of the system in such cases is certainly not optimal.

Simply extending LARA to understand more constructions and words in an attempt to alleviate this problem does not help much because there will always be cases where partial interpretation is the best we can do. Instead, one could implement a feedback component that allows the user to see how the sentence was understood. A simple idea in this direction would be to respond with a yes/no-question that paraphrases what the system has understood. One could try to generate such paraphrases by simply re-linearizing the sentence without the parts that could not be interpreted. For Esperanto, this would work out reasonably well for the instances of partial interpretation discussed in section 5.1.2. For example, a dialogue with Lara could then enclude the following passage:

```
> Krom Esperanton, la homo ankaux lernas cxinan lingvon.

Lara: Bedauxrinde, mi ne komprenis cxion.
Lara: Cxu tio signifas ke homo lernas cxinan lingvon?

> jes.

Okej.
```

Here, the user tells the sytem that "apart from Esperanto, the man also learns Chinese". The system has problems to completely understand this sentence, simply ignores all the dependency nodes it could not interpret, and finally asks the user for feedback by re-linearizing the impoverished structure: "I'm afraid I didn't understand everything. Does that mean that a man learns Chinese ?". The user confirms this, so the system integrates the proposed weak interpretation of the sentence into its model of the situation.

Such a feedback system would certainly not be a very big improvement and would only work well for a few cases, but at least it would facilitate testing the partial interpretation component. For now, however, this must remain an improvement for the future because at the moment the information about the linear order of the dependency nodes gets lost early during semantic processing. Carrying this information over through all of the various stages of semantic processing would first require major changes to the implementation.

7 Conclusion

In this thesis, I have developed and presented a simple constraint system for direct application of classical semantic lambda calculus to dependency structures. In the system, functional applications always happen along dependency links. Only two types of constraints turned out to be necessary: directionality constraints to determine the directionality of functional applications, and precedence constraints to impose restrictions on the order in which dependents are processed.

By using this constraint system to implement an Esperanto version of the CURT dialogue system with about the same syntactic coverage, I have demonstrated that even with such a simple approach, lambda calculus on dependency structures becomes a viable option for deep semantics.

One of the advantages of the system in comparison to rule-based systems is that it allows to define the semantic processing of many different dependency configurations without redundancies.

Apart from this, the constraint system also helps to make semantic interpretation more robust.

Unexpected structures can still be semantically interpreted if functional application along dependency links suffices to derive a valid logical form, which is the case for many structures. This way, the robustness of syntactic dependency parsing can to a certain degree be carried over into the semantic domain.

If a sentence contains problematic words without entries in the semantic lexicon, the method of partial interpretation still allows to retrieve some of the semantic content although some relevant information might be lost. My approach to partial interpretation is to simply ignore such words during semantic processing, which I have shown to yield useful results for quite a few sentences of Esperanto.

As a further step towards robust deep semantics, partial interpretation serves to further increase the coverage of semantic processing. Rather than completely ignore uninterpretable sentences, a system with partial interpretation capabilities will still retrieve some relevant information. This could be useful in the context of semantically interpreting whole texts. However, as I have shown, the loss of information caused by partial interpretation certainly has its risks because some resulting interpretations turn out to be not only weak, but wrong. In the extreme case, the logical interpretation might even be the exact opposite of the user's intention. The severity of this problem is difficult to assess, and quantitative studies to determine the frequency of wrong interpretations are certainly a worthwile project for the future.

In many ways, LARA is not much more than a proof of concept. My straightforward and inefficient implementation of the constraint system has severe performance issues on large structures. LARA is also of limited use for demonstrating the wide-coverage capabilities of my model of robust partial interpretation because the syntactic coverage of the dependency grammar is not very large. The dependency grammar by Schubert (1988) that I started out with allowed wide-coverage parsing of virtually any Esperanto input, but certain non-local constraints in that grammar turned out to be difficult or impossible to express in my DPURC dependency parser. This led to severe overgeneration of the syntactic parsing component and thus made semantic processing far too slow.

As a result, I had to prune the dependency grammar until DPURC only produced a reasonable number of structures for small sentences. After these measures, the syntactic coverage of the grammar that LARA is based on amounts to not much more than the coverage of the CURT system. I still consider this an improvement because the system achieves this coverage for a free word-order language.

Given more efficient implementations of the constraint system and the dependency parser, it would be possible to enlarge the dependency grammar again, making it possible to experiment with wide-coverage semantics on free text input. I have hinted at some ways in which my implementation of the constraint system could be improved, and I hope I will have the opportunity to put these into practice in the future.

8 Bibliography

References

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

Johan Bos. DORIS 2001: Underspecification, Resolution and Inference for Discourse Representation Structures. In *ICoS-3. Inference in Computational Semantics. Workshop Proceedings.*, 2001.

Luísa Coheur, Fernando Batista, and Nuno J. Mamede. Towards a flexible syntax/semantics interface. In *Proceedings of the Herramientas y Recursos Linguísticos para el Español y el Portugués workshop*, pages 256–272, 2004.

James R. Curran, Steven Clark, and Johan Bos. Linguistically Motivated Large-Scale NLP with C&C and Boxer. In *Proceedings of the ACL 2007 Demonstrations Session (ACL-07 demo)*, pages 29–32, 2007.

Johannes Dellert. DPURC - a Dependency Parser with User-definable Relation Constraints. forthcoming.

Iddo Lev, Bill MacCartney, Christopher D. Manning, and Roger Levy. Solving Logic Puzzles: From Robust Processing to Precise Semantics. In *Proceedings of the 2nd Workshop on Text Meaning and Interpretation at ACL 2004*, pages 9–16, 2004.

David Milward. Dynamics, Dependency Grammar and Incremental Interpretation. In *Proceedings of COLING 92*, pages 1095–1099, 1992.

Peter Neuhaus and Norbert Bröker. The Complexity of Recognition of Linguistically Adequate Dependency Grammars. In *Proceedings of the 35th annual meeting of the Association for Computational Linguistics*, pages 337–343, 1997.

Livio Robaldo. *Dependency Tree Semantics.* PhD thesis, University of Turin, Italy, February 2007.

Klaus Schubert. Ausdruckskraft und Regelmäßigkeit: Was Esperanto für automatische Übersetzung geeignet macht. *Language Problems and Language Planning*, 12(2):130–147, 1988.

Klaus Schubert. A dependency syntax of esperanto. In Dan Maxwell and Klaus Schubert, editors, *Metataxis in Practice - Dependency syntax for multilingual machine translation*, pages 207–232. Foris Publications, Dordrecht, Holland, 1989.

John F. Sowa. *Conceptual Structures: Information Processing in Mind and Machine.* Addison-Wesley, Reading, Massachusetts, 1984.

Jan Wielemaker. *SWI-Prolog 5.6 Reference Manual, Updated for version 5.6.59.* University of Amsterdam, August 2008.

L.L Zamenhof [under the pseudonym Ėsperanto]. *Meždunarodnyj jazyk.* Kel'ter, Warsaw, 1887.

9 Appendix

9.1 dpurc-gr-eo.pl

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

    File: dpurc-gr-eo.pl
    Copyright (C) 2008 Johannes Dellert

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

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

/*===========================================================================
   Lexicon for Esperanto (adapted from Schubert 1989)
===========================================================================*/

% PRONOUNS

word(mi, [prn,mi,sg,nom]) :- !.
word(min, [prn,mi,sg,acc]) :- !.
word(vi, [prn,vi,_,nom]) :- !.
word(vin, [prn,vi,_,acc]) :- !.
word(li, [prn,li,sg,nom]) :- !.
word(lin, [prn,li,sg,acc]) :- !.
word(sxi, [prn,sxi,sg,nom]) :- !.
word(sxin, [prn,sxi,sg,acc]) :- !.
word(gxi, [prn,gxi,sg,nom]) :- !.
word(gxin, [prn,gxi,sg,acc]) :- !.
word(ni, [prn,ni,pl,nom]) :- !.
word(nin, [prn,ni,pl,acc]) :- !.
word(ili, [prn,mi,pl,nom]) :- !.
word(ilin, [prn,mi,pl,acc]) :- !.
word(kiu, [prn,kiu,_,nom]) :- !.
word(kiun, [prn,kiu,_,acc]) :- !.
word(tiu, [prn,tiu,_,nom]) :- !.
word(tiun, [prn,tiu,_,acc]) :- !.
word(iu, [prn,iu,sg,nom]) :- !.
word(iun, [prn,iu,sg,acc]) :- !.
word(iuj, [prn,iu,pl,nom]) :- !.
word(iujn, [prn,iu,pl,acc]) :- !.
word(cxiu, [prn,cxiu,sg,nom]) :- !.
word(cxiun, [prn,cxiu,sg,acc]) :- !.
word(cxiuj, [prn,cxiu,pl,nom]) :- !.
word(cxiujn, [prn,cxiu,pl,acc]) :- !.
word(neniu, [prn,neniu,_,nom]) :- !.
word(neniun, [prn,neniu,_,acc]) :- !.
word(kio, [prn,kio,_,nom]) :- !.
word(tio, [prn,tio,_,nom]) :- !.
word(io, [prn,io,_,nom]) :- !.
word(cxio, [prn,cxio,_,nom]) :- !.
word(nenio, [prn,nenio,_,nom]) :- !.
word(kies, [prn,kies,_,nom]) :- !.
word(ties, [prn,ties,_,nom]) :- !.
word(ies, [prn,ies,_,nom]) :- !.
```

```
word(cxies, [prn,cxies,_,nom]) :- !.
word(nenies, [prn,nenies,_,nom]) :- !.
word(ambaux, [prn,ambaux,pl,nom]) :- !.
word(oni,  [prn,oni,sg,nom]).

% PREPOSITIONS

word(al,  [prp,al]) :- !.
word(anstataux,  [prp,anstataux]) :- !.
word(antaux,  [prp,antaux]) :- !.
word(apud,  [prp,apud]) :- !.
word(cxe,  [prp,cxe]) :- !.
word(cxirkaux,  [prp,cxirkaux]) :- !.
word(da,  [prp,da]) :- !.
word(de,  [prp,de]) :- !.
word(dum,  [prp,dum]) :- !.
word(ekster,  [prp,ekster]) :- !.
word(el,  [prp,el]) :- !.
word(en,  [prp,en]) :- !.
word(gxis,  [prp,gxis]) :- !.
word(inter,  [prp,inter]) :- !.
word(je,  [prp,je]) :- !.
word(kontraux,  [prp,kontraux]) :- !.
word(krom,  [prp,krom]) :- !.
word(kun,  [prp,kun]) :- !.
word(kvazaux,  [prp,kvazaux]) :- !.
word(laux, [prp,laux]) :- !.
word(malgraux, [prp,malgraux]) :- !.
word(per,  [prp,per]) :- !.
word(por,  [prp,por]) :- !.
word(post,  [prp,post]) :- !.
word(preter,  [prp,preter]) :- !.
word(pri,  [prp,pri]) :- !.
word(pro,  [prp,pro]) :- !.
word(sen,  [prp,sen]) :- !.
word(sub,  [prp,sub]) :- !.
word(super,  [prp,super]) :- !.
word(sur,  [prp,sur]) :- !.
word(tra,  [prp,tra]) :- !.
word(trans,  [prp,trans]) :- !.

% COORDINATING CONJUNCTIONS BETWEEN VERBS

word(aux,  [konv,aux]).
word(kaj,  [konv,kaj]).
word(nek,  [konv,nek]).
word(sed,  [konv,sed]) :- !.

% COORDINATING CONJUNCTIONS BETWEEN NOUNS

word(aux,  [konn,aux]) :- !.
word(kaj,  [konn,kaj]) :- !.
word(nek,  [konn,nek]) :- !.
```

```
% OTHER COORDINATING CONJUNCTIONS

word(minus,  [kon,minus]) :- !.
word(ol,  [kon,ol]) :- !.
word(plus,  [kon,plus]) :- !.

% SUBORDINATING CONJUNCTIONS

word(cxar, [sbk,cxar]) :- !.
word(cxu, [sbk,cxu]) :- !.
word(do, [sbk,do]) :- !.
word(ke, [sbk,ke]) :- !.
word(kvankam, [sbk,kvankam]) :- !.
word(se, [sbk,se]) :- !.
word(tamen, [sbk,tamen]) :- !.

% INTERJECTIONS

word(nu,  [int,nu]) :- !.

% BOUND FUNCTION MORPHEMES

word(cxi,  [fkc,cxi]) :- !.

% ADVERBS

word(kie, [adv,kie]) :- !.
word(tie, [adv,tie]) :- !.
word(ie, [adv,ie]) :- !.
word(cxie, [adv,cxie]) :- !.
word(nenie, [adv,nenie]) :- !.
word(kial, [adv,kial]) :- !.
word(tial, [adv,tial]) :- !.
word(ial, [adv,ial]) :- !.
word(cxial, [adv,cxial]) :- !.
word(nenial, [adv,nenial]) :- !.
word(kiam, [adv,kiam]) :- !.
word(tiam, [adv,tiam]) :- !.
word(iam, [adv,iam]) :- !.
word(cxiam, [adv,cxiam]) :- !.
word(neniam, [adv,neniam]) :- !.
word(kiel, [adv,kiel]) :- !.
word(tiel, [adv,tiel]) :- !.
word(iel, [adv,iel]) :- !.
word(cxiel, [adv,cxiel]) :- !.
word(neniel, [adv,neniel]) :- !.
word(kiom, [adv,kiom]) :- !.
word(tiom, [adv,tiom]) :- !.
word(iom, [adv,iom]) :- !.
word(cxiom, [adv,cxiom]) :- !.
word(neniom, [adv,neniom]) :- !.
word(almenaux, [adv,almenaux]) :- !.
word(ankaux, [adv,ankaux]) :- !.
word(ankoraux, [adv,ankoraux]) :- !.
```

```
word(apenaux, [adv,apenaux]) :- !.
word(baldaux, [adv,baldaux]) :- !.
word(des, [adv,des]) :- !.
word(ecx, [adv,ecx]) :- !.
word(for, [adv,for]) :- !.
word(hieraux, [adv,hieraux]) :- !.
word(hodiaux, [adv,hodiaux]) :- !.
word(ja, [adv,ja]) :- !.
word(jam, [adv,jam]) :- !.
word(jen, [adv,jen]) :- !.
word(jes, [adv,jes]) :- !.
word(ju, [adv,ju]) :- !.
word(jxus, [adv,jxus]) :- !.
word(mem, [adv,mem]) :- !.
word(morgaux, [adv,morgaux]) :- !.
word(ne,  [adv,ne]) :- !.
word(nun,  [adv,nun]) :- !.
word(nur,  [adv,nur]) :- !.
word(plej,  [adv,plej]) :- !.
word(pli,  [adv,pli]) :- !.
word(plu,  [adv,plu]) :- !.
word(po,  [adv,po]) :- !.
word(preskaux,  [adv,preskaux]) :- !.
word(tre,  [adv,tre]) :- !.
word(tro,  [adv,tro]) :- !.
word(tuj,  [adv,tuj]) :- !.
word(Adverb, [adv,Adverb]) :- sub_atom(Adverb,_,_,0,e).

% NUMERALS

word(nul, [num,nul]) :- !.
word(unu, [num,unu]) :- !.
word(du, [num,du]) :- !.
word(tri, [num,tri]) :- !.
word(kvar, [num,kvar]) :- !.
word(kvin, [num,kvin]) :- !.
word(ses, [num,sep]) :- !.
word(sep, [num,sep]) :- !.
word(ok, [num,ok]) :- !.
word(naux, [num,naux]) :- !.
word(dek, [num,dek]) :- !.
word(cent, [num,cent]) :- !.
word(mil, [num,mil]) :- !.

% ARTICLE

word(la,  [art,la]) :- !.
```

```
% NOUNS (open word class defined by morphology)

word(Noun, [sub,Noun,sg,nom]) :- sub_atom(Noun,_,_,0,o).

word(Noun, [sub,Lemma,sg,acc]) :-
    sub_atom(Noun,Index,_,0,on),
    CutIndex is Index + 1 ,
    sub_atom(Noun,0,CutIndex,_,Lemma).

word(Noun, [sub,Lemma,pl,nom]) :-
    sub_atom(Noun,Index,_,0,oj),
    CutIndex is Index + 1 ,
    sub_atom(Noun,0,CutIndex,_,Lemma).

word(Noun, [sub,Lemma,pl,acc]) :-
    sub_atom(Noun,Index,_,0,ojn),
    CutIndex is Index + 1 ,
    sub_atom(Noun,0,CutIndex,_,Lemma).


% ADJECTIVES (open word class defined by morphology)

word(Adjective, [adj,Adjective,sg,nom]) :- sub_atom(Adjective,_,_,0,a).

word(Adjective, [adj,Lemma,sg,acc]) :-
    sub_atom(Adjective,Index,_,0,an),
    CutIndex is Index + 1 ,
    sub_atom(Adjective,0,CutIndex,_,Lemma).

word(Adjective, [adj,Lemma,pl,nom]) :-
    sub_atom(Adjective,Index,_,0,aj),
    CutIndex is Index + 1 ,
    sub_atom(Adjective,0,CutIndex,_,Lemma).

word(Adjective, [adj,Lemma,pl,acc]) :-
    sub_atom(Adjective,Index,_,0,ajn),
    CutIndex is Index + 1 ,
    sub_atom(Adjective,0,CutIndex,_,Lemma).


% VERBS (open word class defined by morphology)

word(Verb, [vrb,Verb,infin]) :- sub_atom(Verb,_,_,0,i).

word(Verb, [vrb,Lemma,pres]) :-
    sub_atom(Verb,Index,_,0,as),
    sub_atom(Verb,0,Index,_,Stem),
    atom_concat(Stem,i,Lemma).

word(Verb, [vrb,Lemma,pret]) :-
    sub_atom(Verb,Index,_,0,is),
    sub_atom(Verb,0,Index,_,Stem),
    atom_concat(Stem,i,Lemma).
```

```
word(Verb, [vrb,Lemma,fut]) :-
    sub_atom(Verb,Index,_,0,os),
    sub_atom(Verb,0,Index,_,Stem),
    atom_concat(Stem,i,Lemma).

word(Verb, [vrb,Lemma,cond]) :-
    sub_atom(Verb,Index,_,0,us),
    sub_atom(Verb,0,Index,_,Stem),
    atom_concat(Stem,i,Lemma).

word(Verb, [vrb,Lemma,imp]) :-
    sub_atom(Verb,Index,_,0,u),
    sub_atom(Verb,0,Index,_,Stem),
    atom_concat(Stem,i,Lemma).


/*=========================================================================
    Dependency Rules for Esperanto (adapted from Schubert 1989)

    Each rule is of the form
    dh(Dependent,Head,Headedness,Projectivity,Uniqueness,Label) where

    Dependent and Head are like the lexical entries above,
    Headedness has one of the values free_order, head_initial or head_final,
    Projectivity has one of the values yes or no,
    Uniqueness has one of the values yes or no, and
    Label is a label for the relation established.
=========================================================================*/

%% VERB as governor

% Noun as subject of a verb
% --> Fero rustigxas.
dh([sub,_,_,nom],[vrb,_,_],free_order,yes,yes,subj).

% Pronoun as subject of a verb
% --> Gxi kolektigxis.
dh([prn,_,_,nom],[vrb,_,_],free_order,yes,yes,subj).

% Noun as object of a verb
% --> Li acxetis auxton.
dh([sub,_,_,acc],[vrb,_,_],free_order,yes,yes,obj).

% Pronoun as object of a verb
% --> Li acxetis gxin.
dh([prn,_,_,acc],[vrb,_,_],free_order,yes,yes,obj).

% Noun as predicative of "esti"
% --> Li estas prezidanto.
dh([sub,_,_,nom],[vrb,esti,_],free_order,yes,yes,pred).

% Pronoun as predicative of "esti"
% --> Tio estas gxi.
dh([prn,_,_,nom],[vrb,esti,_],free_order,yes,yes,pred).
```

```
% Adjective as predicative of "esti"
% --> Gxi estas blua.
dh([adj,_,_,nom],[vrb,esti,_],free_order,yes,yes,pred).

% Preposition as propositional argument of a verb
% --> Li dormis en sia lito.
dh([prp,_],[vrb,_,_],free_order,yes,yes,proa).

% Subordinating conjunction as propositional argument of a verb
% --> Li ne povis veni, do li devis malaligxi.
dh([sbk,_],[vrb,_,_],free_order,yes,yes,proa).

% Adverb as modal adjunct of a verb
% --> Pove, mi ne venos al vi.
dh([adv,_],[vrb,_,_],free_order,yes,yes,moda).


%% NOUN as governor

% Article as determiner of a noun
% --> la domo
dh([art,_],[sub,_,_,_],head_final,yes,yes,det).

% Pronoun as attribute of a noun
% --> iu domo
dh([prn,_,_,Case],[sub,_,_,Case],head_final,yes,yes,atr1).

% Adjective as attribute of a noun
% --> blua domo
dh([adj,_,Number,Case],[sub,_,Number,Case],free_order,yes,no,atr1).

% Numeral as attribute of a noun
% --> unu domo
dh([num,_],[sub,_,_,_],free_order,yes,yes,atr1).

% Preposition as free adjunct of a noun
% --> la auxto de mia amiko
dh([prp,_],[sub,_,_,_],head_initial,yes,yes,adju).

% Verb as relative clause of a noun
% --> domo, kie ni logxas
dh([vrb,_,_],[sub,_,_,_],head_initial,yes,yes,rel).


%% PRONOUN as governor

% Pronoun as attribute of another pronoun
% --> Ili renkontas nin cxiujn.
dh([prn,_,Number,_],[prn,_,Number,_],free_order,yes,yes,atr1).

% Numeral as apposition of a pronoun
% --> Tri iuj
dh([num,_],[prn,_,_,_],head_final,yes,yes,apo).
```

```
% Adverb as adjunct of a pronoun
% --> ni cxi tie
dh([adv,_],[prn,_,_,_],head_initial,yes,yes,adju).

% Preposition as adjunct of a pronoun
% --> iu el miaj amikoj
dh([prp,_],[prn,_,_,_],head_initial,yes,yes,adju).

%% PREPOSITION as governor

% Noun in argument position (only accepts nominative case!)
% --> pri domo
dh([sub,_,_,nom],[prp,_],head_initial,yes,yes,parg).

%% SUBORDINATING CONJUNCTION as governor

% Verb as head of the subordinated clause
% --> Mi venos, se mi ricevos bileton.
dh([vrb,_,_],[sbk,_],head_initial,yes,yes,subc).

% Verbal coordination as head of the subordinated clause
% --> Mi venos, se hundo dormos kaj ne mordos min.
dh([konv,_],[sbk,_],head_initial,yes,yes,subc).

%% VERBAL COORDINATION as governor

% Noun as subject of a verbal coordination
% --> Fero rustigxas kaj estis for.
dh([sub,_,_,nom],[konv,_],free_order,yes,yes,subj).

% Pronoun as subject of a verbal coordination
% --> Gxi kolektigxis kaj iris for.
dh([prn,_,_,nom],[konv,_],free_order,yes,yes,subj).

% Noun as object of a verbal coordination
% --> Li acxetis kaj vendis auxtojn.
dh([sub,_,_,acc],[konv,_],free_order,yes,yes,obj).

% Pronoun as object of a verbal coordination
% --> Li acxetis kaj mangxis gxin.
dh([prn,_,_,acc],[konv,_],free_order,yes,yes,obj).

% Preposition as propositional argument of a verbal coordination
% --> Li dormis kaj vivis en sia lito.
dh([prp,_],[konv,_],free_order,yes,yes,proa).

% Verb as left coordinate of a verbal coordination
% --> Ni esperis kaj atendis.
dh([vrb,_,_],[konv,_],head_final,yes,yes,konv1).

% Verb as right coordinate of a verbal coordination
% --> Ni esperis kaj atendis.
dh([vrb,_,_],[konv,_],head_initial,yes,yes,konv2).
```

9.2 eo-dict-sem.pl

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

    File: eo-dict-sem.pl
    Copyright (C) 2008 Johannes Dellert

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

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

/*===========================================================================
   Semantic Lexicon for Esperanto
===========================================================================*/

% Complex lexical rule describing the different interpretations for nouns

semLex(sub,Lemma,Sem):-
   % Standard Noun
   (
    Sem = lam(X,Formula),
    compose(Formula,Lemma,[X])
   );
   % Proper Nouns (syntactic parser can't distinguish them from nouns!)
   (
    Sem = lam(P,app(P,Lemma))
   ), !.


% Copula

semLex(vrb, esti, Sem) :-
  Sem = lam(K,lam(Y,app(K,lam(X,eq(Y,X))))), !.


% Complex lexical rule for verbs that can either be transitive or intransitive

semLex(vrb,Lemma,Sem):-
  % Intransitive Verb
  (
    Sem = lam(X,Formula),
    compose(Formula,Lemma,[X])
  );
  % Transitive Verb
  (
    Sem = lam(K,lam(Y,app(K,lam(X,Formula)))),
    compose(Formula,Lemma,[Y,X])
  ), !.


% Negation

semLex(adv,ne,Sem) :-
  Sem = lam(P,lam(X,not(app(P,X)))),
  Sem = lam(P,not(P)), !.
```

```
% Universal Quantification

semLex(prn, cxiu, Sem) :-
  Sem = lam(U,lam(V,all(X,imp(app(U,X),app(V,X))))), !.

% Existential Quantification (usually implicit)

semLex(prn, iu, Sem) :-
  Sem = lam(U,lam(V,some(X,and(app(U,X),app(V,X))))), !.

% Relative / Interrogative pronoun "kiu"

semLex(prn, kiu , Sem):-
   Sem = lam(P,lam(Q,lam(X,and(app(P,X),app(Q,X))))));
   Sem = lam(Q,que(X,X,app(Q,X))), !.

%  Relative / Interrogative pronoun "kio"

semLex(prn, kio , Sem):-
   Sem = lam(P,lam(Q,lam(X,and(app(P,X),app(Q,X))))),
   Sem = lam(Q,que(X,X,app(Q,X))), !.

% Other pronouns simply treated as constants

semLex(prn, Lemma, Sem) :-
  Sem = lam(P,app(P,Lemma)), !.

% Adjectives

semLex(adj, Lemma, Sem) :-
  Sem = lam(P,lam(X,and(F,app(P,X)))),
  compose(F,Lemma,[X]), !.

% Prepositions

semLex(prp, Lemma, Sem) :-
   Sem = lam(K,lam(P,lam(Y,and(app(K,lam(X,F)),app(P,Y))))),
   compose(F,Lemma,[Y,X]), !.

% Verbal coordination

semLex(konv, kaj, Sem) :-
  Sem = lam(X,lam(Y,lam(P,lam(Q,and(app(app(X,Q),P),app(app(Y,Q),P)))))));
  Sem = lam(X,lam(Y,lam(P,and(app(X,P),app(Y,P))))), !.

semLex(konv, aux, Sem) :-
  Sem = lam(X,lam(Y,lam(P,lam(Q,or(app(app(X,Q),P),app(app(Y,Q),P)))))));
  Sem = lam(X,lam(Y,lam(P,or(app(X,P),app(Y,P))))), !.

% Subordinating conjunctions

semLex(sbk, se, Sem) :-
  Sem = lam(X,lam(Y,imp(X,Y))), !.
```

```
% default for uninterpretable tokens: unknown semantics
% is processed even if there are viable alternatives if these don't end with a cut!

semLex(_,_,unknown).

/*=========================================================================
   Directionality Constraints for Esperanto
    -  specify in which direction heads and dependents are combined
    -  enriched by additional instructions for implicit quantification
=========================================================================*/

directional(vrb:A,  subj,   sub:B,  app(B,A)).
directional(vrb:A,  obj,    sub:B,  app(A,B)).
directional(vrb:A,  pred,   sub:B,  app(A,B)).
directional(konv:A, subj,   sub:B,  app(B,A)).
directional(konv:A,  obj,    sub:B,  app(B,A)).

% implicit existential quantification
directional(vrb:A,  subj,   sub:B,
  app(app(lam(U,lam(V,some(X,and(app(U,X),app(V,X))))),B),A)) :- !.
directional(vrb:A,  obj,    sub:B,
  app(A,app(lam(U,lam(V,some(X,and(app(U,X),app(V,X))))),B))) :- !.
directional(vrb:A,  pred,   sub:B,
  app(A,app(lam(U,lam(V,some(X,and(app(U,X),app(V,X))))),B))) :- !.
directional(konv:A, subj,   sub:B,
  app(app(lam(U,lam(V,some(X,and(app(U,X),app(V,X))))),B),A)) :- !.

directional(vrb:A,  proa,   prp:B,  app(B,A)) :- !.
directional(vrb:A,  proa,   sbk:B,  app(B,A)) :- !.
directional(vrb:A,  moda,   adv:B,  app(B,A)) :- !.
directional(sub:A,  det,    prn:B,  app(B,A)) :- !.
directional(sub:A,  atr1,   adj:B,  app(B,A)) :- !.
directional(sub:A,  atr1,   prn:B,  app(B,A)) :- !.
directional(sub:A,  rel,    vrb:B,  app(B,A)) :- !.
directional(prp:A,  parg,   sub:B,  app(A,B)) :- !.
directional(sbk:A,  subc,   vrb:B,  app(A,B)) :- !.

directional(_:A,    _,      _:B,    app(B,A)).
directional(_:A,    _,      _:B,    app(A,B)).

/*=========================================================================
   Precedence Constraints for Esperanto
    - specify in which order dependents are combined with their head
=========================================================================*/

precedence(vrb,   obj,    subj).
precedence(vrb,   pred,   subj).
precedence(vrb,   subj,   proa).
precedence(sub,   atr1,   det).
precedence(konv,  konv1,  konv2).
precedence(konv,  konv2,  obj).
precedence(konv,  obj,    subj).
```

9.3 dpurc-sem.pl

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

    File: dpurc-sem.pl
    Copyright (C) 2008 Johannes Dellert

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

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

:- use_module(comsemPredicates,[infix/0,
                                prefix/0,
                                printRepresentations/1,
                                memberList/2,
                                reverseList/2,
                                selectFromList/3,
compose/3]).

:- use_module(betaConversion,[betaConvert/2]).

:- use_module(alphaConversion,[alphabeticVariants/2]).

:- [dpurc].

:- [dpurc-gr-eo].

:- [eo-dict-sem].

/*===========================================================================
   Retrieve all the LFs for an input sentence
===========================================================================*/

syn_sem_parse(Input, Sems) :-
  setof(Sem,syn_sem_single_parse(Input,Sem),AllSems),
  filterAlphabeticVariants(AllSems,Sems).


/*===========================================================================
   Retrieve a single LF for an input sentence
===========================================================================*/

syn_sem_single_parse(Input, Sem) :-
  parse(Input,[Head]),
  sem_single_parse(Head,Sem).


/*===========================================================================
   Retrieve all the LFs for a dependency structure
===========================================================================*/

sem_parse(Head,Sems) :-
  setof(Sem,sem_single_parse(Head,Sem),AllSems),
  filterAlphabeticVariants(AllSems,Sems).
```

```
/*========================================================================
   Retrieve a single LF for a dependency structure
========================================================================*/

sem_single_parse(Head, Converted) :-
  sem_proc(Head, Sem),
  betaConvert(Sem,Converted),
  % suppress non-sentential LFs
  \+ lambda_or_app_occurrence(Converted) .


/*========================================================================
   Compute LFs for a dependency structure
========================================================================*/

sem_proc([_,_,_,_,_,_,Dependents,_,Cat,Lemma|_], Sem) :-
  Dependents = [],
  semLex(Cat, Lemma, Sem).

sem_proc([_,_,_,_,_,_,Dependents,_,Cat,Lemma|_],Sem) :-
  compute_dep_sem(Dependents, [], DepSems),
  permutation(DepSems, InverseDepSemsVariant),
  %must be checked in inverse order
  \+ violates_precedence(Cat,InverseDepSemsVariant),
  reverseList(InverseDepSemsVariant, DepSemsVariant),
  semLex(Cat,Lemma,HeadSem),
  apply_directionally(Cat, HeadSem, DepSemsVariant, Sem).


/*========================================================================
   Compute semantic values for dependents
========================================================================*/

compute_dep_sem([],DepSems,DepSems).

compute_dep_sem([Dep|Dependents], DepSems, ReturnDepSems) :-
  % check necessary because Dependents is an open list; also: base case
  \+ var(Dep),
  sem_proc(Dep, DepSem),
  Dep = [Label|_],
  append(DepSems,[Label:DepSem],NewDepSems),
  compute_dep_sem(Dependents, NewDepSems, ReturnDepSems).


/*========================================================================
   Find precedence constraint violations in (inverse) orders of dependent LFs
========================================================================*/

violates_precedence(HeadCat,[First:_|DepSemsSequence]) :-
  ( precedence(HeadCat,First,Preceded),
    memberList(DepSemsSequence,Preceded:_));
  violates_precedence(HeadCat,DepSemsSequence).
```

```
/*====================================================================
   Combine semantics of a head and its dependents
   according to the directionality constraints
====================================================================*/

apply_directionally(_,HeadSem,[],HeadSem).

apply_directionally(_,unknown,_,unknown).

apply_directionally(HeadCat,HeadSem,[_:unknown|OtherDepSems],FinalSem) :-
  apply_directionally(HeadCat,HeadSem,OtherDepSems,FinalSem).

apply_directionally(HeadCat,HeadSem,[Label:DepSem|OtherDepSems],FinalSem) :-
  directional(HeadCat:HeadSem,Label,_:DepSem,NextSem),
  apply_directionally(HeadCat,NextSem,OtherDepSems,FinalSem).


/*====================================================================
   Helper predicates to generate all permutations
====================================================================*/

select(X,[X|Xs],Xs).
select(X,[Y|Ys],[Y|Zs]) :- select(X,Ys,Zs).

permutation(Xs,[Z|Zs]) :- select(Z,Xs,Ys), permutation(Ys,Zs).
permutation([],[]).


/*====================================================================
   Find non-sentential LFs in B&B notation
====================================================================*/

lambda_or_app_occurrence(Term) :-
  \+ var(Term),
  functor(Term,Name,Arity),
  (
    Name = lam;
    Name = app;
    (
      arg(_,Term,_),
      lambda_or_app_occurrence_args(Term, 1, Arity)
    )
  ).

lambda_or_app_occurrence_args(Term, ArgID, Arity) :-
  arg(ArgID,Term,Arg),
  (
    lambda_or_app_occurrence(Arg);
    (
      NextArgID is ArgID + 1,
      NextArgID =< Arity,
      lambda_or_app_occurrence_args(Term,NextArgID, Arity)
    )
  ).
```

```
/*========================================================================
   Filter Alphabetic Variants (Copyright Blackburn & Bos)
========================================================================*/

filterAlphabeticVariants(L1,L2):-
   selectFromList(X,L1,L3),
   memberList(Y,L3),
   alphabeticVariants(X,Y), !,
   filterAlphabeticVariants(L3,L2).

filterAlphabeticVariants(L,L).


/*========================================================================
   Collect and print all the LFs for a sentence
========================================================================*/

try_all_sem(List) :-
  syn_sem_parse(List, Sems),
  printRepresentations(Sems).


/*========================================================================
   Collect and print all the dependency structures for a sentence
   together with their interpretations (if any)
========================================================================*/

try_sem(List) :-
  write_list(List),nl,
  parse(List,[Head]),
  write_dep(Head),
  sem_parse(Head, Formulas),
  printRepresentations(Formulas),
nl,
  nl,
fail.

try_sem(_) :- write('No (more) parses.'), nl.
```

9.4 laraPredicates.pl

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

    File: laraPredicates.pl
    Copyright (C) 2008 Johannes Dellert

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

    Original file: curtPredicates.pl
    Copyright (C) 2004,2005,2006 Patrick Blackburn & Johan Bos

    That file was part of BB1, version 1.3 (November 2006).

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

:- module(laraPredicates,[laraHelp/0,
                          laraOutput/1,
                          updateReadings/1,
                          updateModels/1,
                          updateHistory/1,
                          clearHistory/0,
                          list2string/2,
                          selectReadings/3]).

:- use_module(comsemPredicates,[appendLists/3]).

/*============================================================================
   Lara Help
============================================================================*/

laraHelp:-
   nl, write('interpretoj: printas la interpretojn aktualajn'),
   nl, write('elektu N: elektu interpreton (N devas esti entjero)'),
   nl, write('denove: iniciatas novan diskurson'),
   nl, write('historio: montras la diskurson gxis nun'),
   nl, write('modeloj: printas la modelojn aktualajn'),
   nl, write('resumo: eliminas interpretojn ekvivalentajn'),
   nl, write('scio: kalkulas kaj montras fonan scion'),
   nl, write('infikso: vidigu formulojn en infiksa notado'),
   nl, write('prefikso: vidigu formulojn en prefiksa notado'),
   nl, write('gxis: forlasi Laran'),
   nl.


/*============================================================================
   Lara's output
============================================================================*/

laraOutput([]).

laraOutput([Move|Moves]):-
   realiseMove(Move,Output),
   format('~nLara: ~p~n',[Output]),
   laraOutput(Moves).
```

```
/*=========================================================================
   Lara's Moves
=========================================================================*/

realiseMove(clarify,'Cxu vi volas diri ion?').
realiseMove(bye,'Gxis revido!').
realiseMove(accept,'Okej.').
realiseMove(noparse,'Kio?').
realiseMove(contradiction,'Ne! Mi ne kredas tion!').
realiseMove(obvious,'Nu, tio estas evidenta!').
realiseMove(unknown_answer,'Mi ne havas ideon.').
realiseMove(sensible_question,'Senca demando!').
realiseMove(answer(String),String).


/*=========================================================================
   Select Readings
=========================================================================*/

selectReadings(X,R1,R2):-
   selectReadings(1,X,R1,R2).

selectReadings(X,X,[R|_],[R]).

selectReadings(X,Y,[_|L],R):-
   X < Y,
   Z is X + 1,
   selectReadings(Z,Y,L,R).


/*=========================================================================
   Update History
=========================================================================*/

updateHistory(Input):-
   retract(lara:history(His1)),
   appendLists(His1,[Input],His2),
   assert(lara:history(His2)).


/*=========================================================================
   Clear History
=========================================================================*/

clearHistory:-
   retract(lara:history(_)),
   assert(lara:history([])).
```

```
/*=========================================================================
   Update Readings
=========================================================================*/

updateReadings(R):-
   retract(lara:readings(_)),
   assert(lara:readings(R)).


/*=========================================================================
   Update Models
=========================================================================*/

updateModels(R):-
   retract(lara:models(_)),
   assert(lara:models(R)).


/*=========================================================================
    Convert a list of words to a string
=========================================================================*/

list2string([Word],Word).

list2string([Word|L],String2):-
   list2string(L,String1),
   name(Word,Codes1),
   name(String1,Codes2),
   appendLists(Codes1,[32|Codes2],Codes3),
   name(String2,Codes3).
```

9.5 helpemaLara.pl

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

    File: helpemaLara.pl
    Copyright (C) 2008 Johannes Dellert

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

    Original file: helpfulCurt.pl
    Copyright (C) 2004,2005,2006 Patrick Blackburn & Johan Bos

    That file was part of BB1, version 1.3 (November 2006).

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

:- module(lara,[lara/0,infix/0,prefix/0]).

:- use_module(callInference,[callTP/3,
                             callTPandMB/6]).

:- use_module(readLine,[readLine/1]).

:- use_module(comsemPredicates,[infix/0,
                                prefix/0,
                                memberList/2,
                                compose/3,
                selectFromList/3,
                printRepresentations/1]).

:- use_module(modelChecker2,[satisfy/4]).

:- use_module(backgroundKnowledge,[backgroundKnowledge/2]).

:- use_module(elimEquivReadings,[elimEquivReadings/2]).

:- use_module(laraPredicates,[laraHelp/0,
                              laraOutput/1,
                              updateReadings/1,
                              updateModels/1,
                              updateHistory/1,
                              clearHistory/0,
                              list2string/2,
                              selectReadings/3]).
:- [dpurc-sem].

/*============================================================================
   Dynamic Predicates
============================================================================*/

:- dynamic history/1, readings/1, models/1.

history([]).
readings([]).
models([]).
```

```
/*=========================================================================
   Start Lara
=========================================================================*/

lara:-
   laraTalk(run).

/*=========================================================================
   Control
=========================================================================*/

laraTalk(quit).

laraTalk(run):-
   readLine(Input),
   laraUpdate(Input,LarasMoves,State),
   laraOutput(LarasMoves),
   laraTalk(State).


/*=========================================================================
   Update Lara's Information State
=========================================================================*/

laraUpdate([],[clarify],run):- !.

laraUpdate([gxis],[gxis],quit):- !,
   updateReadings([]),
   updateModels([]),
   clearHistory.

laraUpdate([denove],[],run):- !,
   updateReadings([]),
   updateModels([]),
   clearHistory.

laraUpdate([helpo],[],run):- !,
   laraHelp.

laraUpdate([infikso],[],run):- !,
   infix.

laraUpdate([prefikso],[],run):- !,
   prefix.

laraUpdate([elektu,X],[],run):-
   number(X),
   readings(R1),
   selectReadings(X,R1,R2), !,
   updateReadings(R2),
   models(M1),
   selectReadings(X,M1,M2),
   updateModels(M2).
```

```
laraUpdate([resumo],[],run):-
   readings(Readings),
   elimEquivReadings(Readings,Unique),
   updateReadings(Unique),
   updateModels([]).

laraUpdate([scio],[],run):-
   readings(R),
   findall(K,(memberList(F,R),backgroundKnowledge(F,K)),L),
   printRepresentations(L).

laraUpdate([interpretoj],[],run):- !,
   readings(R),
   printRepresentations(R).

laraUpdate([modeloj],[],run):- !,
   models(M),
   printRepresentations(M).

laraUpdate([historio],[],run):- !,
   history(H),
   printRepresentations(H).

laraUpdate(Input,Moves,run):-
   syn_sem_parse(Input,Readings), !,
   updateHistory(Input),
   (
      Readings=[que(X,R,S)|_],
      models(OldModels),
      answerQuestion(que(X,R,S),OldModels,Moves)
   ;
      \+ Readings=[que(_,_,_)|_],
      consistentReadings(Readings,[]-ConsReadings,[]-Models),
      (
         ConsReadings=[],
         Moves=[contradiction]
      ;
         \+ ConsReadings=[],
         informativeReadings(ConsReadings,[]-InfReadings),
         (
            InfReadings=[],
            Moves=[obvious]
         ;
            \+ InfReadings=[],
            Moves=[accept]
         ),
         combine(ConsReadings,CombinedReadings),
         updateReadings(CombinedReadings),
         updateModels(Models)
      )
   ).

laraUpdate(_,[noparse],run).
```

```
/*============================================================================
   Combine New Utterances with History
============================================================================*/

combine(New,New):-
   readings([]).

combine(Readings,Updated):-
   readings([Old|_]),
   findall(and(Old,New),memberList(New,Readings),Updated).


/*============================================================================
   Select Consistent Readings
============================================================================*/

consistentReadings([],C-C,M-M).

consistentReadings([New|Readings],C1-C2,M1-M2):-
   readings(Old),
   (
      consistent(Old,New,Model), !,
      consistentReadings(Readings,[New|C1]-C2,[Model|M1]-M2)
   ;
      consistentReadings(Readings,C1-C2,M1-M2)
   ).


/*============================================================================
   Consistency Checking calling Theorem Prover and Model Builder
============================================================================*/

consistent([Old|_],New,Model):-
   DomainSize=15,
   callTPandMB(not(and(Old,New)),and(Old,New),DomainSize,Proof,Model,_),
   \+ Proof=proof, Model=model([_|_],_).

consistent([],New,Model):-
   DomainSize=15,
   callTPandMB(not(New),New,DomainSize,Proof,Model,_),
   \+ Proof=proof, Model=model([_|_],_).
```

```
/*========================================================================
   Select Informative Readings
========================================================================*/

informativeReadings([],I-I).

informativeReadings([New|L],I1-I2):-
   readings(Old),
   (
      informative(Old,New), !,
      informativeReadings(L,[New|I1]-I2)
   ;
      informativeReadings(L,I1-I2)
   ).


/*========================================================================
   Informativity Checking calling Theorem Prover
========================================================================*/

informative([Old|_],New):-
   DomainSize=15,
    callTPandMB(not(and(Old,not(New))),and(Old,not(New)),DomainSize,Proof,Model,_),
   \+ Proof=proof, Model=model([_|_],_).

informative([],New):-
   DomainSize=15,
    callTPandMB(not(not(New)),not(New),DomainSize,Proof,Model,_),
   \+ Proof=proof, Model=model([_|_],_).


/*========================================================================
    Answer Questions
========================================================================*/

answerQuestion(que(X,R,S),Models,Moves):-
   (
      Models=[Model|_],
      satisfy(some(X,S),Model,[],Result),
      \+ Result=undef,
      !,
      findall(A,satisfy(S,Model,[g(X,A)],pos),Answers),
      realiseAnswer(Answers,que(X,R,S),Model,String),
      Moves=[sensible_question,answer(String)]
   ;
      Moves=[unknown_answer]
   ).
```

```
/*=========================================================================
    Realise all answers
=========================================================================*/

realiseAnswer([],_,_,'neniu').

realiseAnswer([Value],Q,Model,String):-
   realiseString(Q,Value,Model,String).

realiseAnswer([Value1,Value2|Values],Q,Model,String):-
   realiseString(Q,Value1,Model,String1),
   realiseAnswer([Value2|Values],Q,Model,String2),
   list2string([String1,kaj,String2],String).


/*=========================================================================
    Realise a single answer
=========================================================================*/

realiseString(que(X,R,S),Value,Model,String):-
   Model = model(_,RelList),
   memberList(Rel,RelList),
   Rel = f(0,Symbol,Value),
   satisfy(eq(Y,Symbol),Model,[g(Y,Value)],pos), !,
   checkAnswer(some(X,and(eq(X,Symbol),and(R,S))),Proof),
   (
      Proof=proof, !,
      list2string([Symbol],String)
   ;
      list2string([eble|Symbol],String)
   ).

realiseString(que(X,R,S),Value,Model,String):-
   Model = model(_,RelList),
   memberList(Rel,RelList),
   Rel = f(1,Symbol,Set),
   memberList(Value,Set),
   compose(Formula,Symbol,[X]),
   satisfy(Formula,Model,[g(X,Value)],pos), !,
   checkAnswer(some(X,and(Formula,and(R,S))),Proof),
   (
      Proof=proof, !,
      list2string([Symbol],String)
   ;
      list2string([eble|Symbol],String)
   ).

realiseString(_,Value,_,Value).
```

```
/*============================================================================
   Answer Checking
============================================================================*/

checkAnswer(Answer,Proof):-
   readings([F|_]),
   backgroundKnowledge(F,BK),
   callTP(imp(and(F,BK),Answer),Proof,_).


/*============================================================================
   Info
============================================================================*/

info:-
   format('~n> ------------------------------------------------------- <',[]),
   format('~n> helpemaLara.pl, de Johannes Dellert                      <',[]),
   format('~n>                                                          <',[]),
   format('~n> ?- lara.                - iniciatu dialogon kun Lara     <',[]),
   format('~n>                                                          <',[]),
   format('~n> Tajpu "helpo" por pli informo pri la trajtoj de Lara     <',[]),
   format('~n> ------------------------------------------------------- <',[]),
   format('~n~n',[]).



/*============================================================================
   Display info at start
============================================================================*/

:- info.
```