Translating Natural Language Sentences into ASP theories using SE-DCG grammars and Lambda Calculus

Stefania Costantini and Alessio Paolucci

Dip. di Informatica, Università di L'Aquila, Coppito 67100, L'Aquila, Italy
stefcost@di.univaq.it, alessiopaolucci@ieee.org

Abstract. We build upon recent work by Baral, Dzifcal and Son that define the translation into ASP of (some classes of) natural language sentences from the lambda-calculus intermediate format generated by CCG grammars. We propose to use SE-DCG grammars, and we introduce automatic generation of lambda-calculus expressions from template ones, thus improving the effectiveness and generality of the translation process.

1 Introduction

Many intelligent systems have to deal with knowledge expressed in natural language, either extracted from books, web pages and documents in general, or expressed by human users. Knowledge acquisition from these sources is a challenging matter, and many attempts are presently under way towards automatically translating natural language sentences into an appropriate knowledge representation formalism [1]. The selection of a suitable formalism plays an important role but first-order logic, that would under many respects represent a natural choice, is actually not appropriate for expressing various kinds of knowledge, i.e., for dealing with default statements, normative statements with exceptions, etc. Recent work has investigated the usability of non-monotonic logics, like Answer Set Programming (ASP)[2].

The so-called Web 3.0 [3][4], despite its definition not well-established yet, makes the important assumption that applications should accept knowledge expressed in a human-like form, transform it into a machine processable form and take this step as the basis for semantic applications. This bears a similarity with the Semantic Web objectives [4], though Web 3.0 is a much wider vision, where artificial intelligence techniques plays a central role. Also in the semantic web scenario however, automatically extracting semantic information from web pages or text documents requires to deal with natural language processing, and requires forms of reasoning.

Translating natural language sentences into a logic knowledge representation is a key point on the applications side as well. In fact, designing applications such as semantic search engines implies obtaining a machine-processable form of the extracted knowledge that makes it possible to perform reasoning on the data so

as to suitably answer (possibly in natural language) to the user's queries, as such an engine should interact with the user like a personal agent. We have practically demonstrated in previous work [5] that for extracting semantic information from a large dataset like wikipedia, a reasoning process on the data is needed, e.g., for semantic disambiguation of concepts.

A central aspect of knowledge acquisition is related to the automation of the process. Recent work in this direction has been presented in [1] [6] and [2]. In our opinion, the latter represents a significant advancement towards automatic translation of natural language sentences into a knowledge representation format that allows for automated reasoning. This works outlines a method for translating natural language sentences into ASP, so as to be able to reason on the extracted knowledge. They try in particular to take into account sentences defining uncertain and defeasible knowledge. In this paper, we extend their approach by adopting a semantically enhanced efficient context-free parser and introducing a new more abstract intermediate representation to be instantiated on practical cases. Also, we propose a fully automated translation methodology based upon our previous work [5].

2 Background

Before entering into the details of our proposal, we need to introduce the necessary building blocks of the work of [2]. They use CCG grammars that produce a λ-calculus intermediate form of a given sentences. Then, they introduce a variant of this intermediate form so as to cope with uncertain knowledge, and finally they propose a translation into ASP. Below we shortly recall the basics of λ-calculus and CCG's, and then illustrate the method of [2].

2.1 Lambda Calculus

λ-calculus is a formal system designed to investigate function definition, function application and recursion. Alonzo Church and Stephen Cole Kleene introduced λ-calculus in the 1930s as part of an investigation into the foundations of mathematics, but it can be seen as programming language: in fact, it is universal in the sense that any computable function can be expressed and evaluated via this formalism.

The central concept in λ calculus is the "expression", defined recursively as follows (where a "variable", is an identifier which can be any of the letters a, b, c, ...):

$$M ::= < name > | < function > | < application >$$
$$< function > ::= \lambda < name > .M$$
$$< application > ::= MM$$

Parentheses can be used for clarity. λ-calculus has only two keywords: λ and the dot. A single identifier is a valid λ expression, like, e.g., $\lambda x.x$ that defines the identity function. The name after the λ is the identifier of the argument of

this function. The expression after the point is called the *body* of the definition. Functions can be applied to expressions, like, e.g., $(\lambda x.x)y$ which is the identity function applied to y. Parentheses are used to avoid ambiguity. Function applications are evaluated by substituting the value of the argument x (in this case 'y') in the body of the function definition. The names of the arguments in function definitions do not carry any meaning by themselves, they are just place holders. In λ-calculus all names are local to definitions. In the function $\lambda x.x$, x is bound since its occurrence in the body of the definition is preceded by λx. A name not preceded by a λ is called a free variable. The same identifier can occur free and bound in the same expression.

Substitution corresponds to the operation that will replace in a term all the free occurrences of x with y, like $[y/x]M$

An α-conversion allows bounded variables to change their name, like $\lambda x.M = \lambda y.[y/x]M$ where $[y/x]M$ is the result of substituting y for free occurrences of x in M and y cannot already appear in M.

β-reduction defines an axiom related to the idea of function application. The β-reduction of $((\lambda x.M)N)$ is $[N/x]M$ where $[N/x]M$ denotes the substitution of the formal parameter x with the argument N throughout the expression M. In the rest of this paper, β-reduction will be denoted by the connective @, where for example $\lambda x.flies(x)\,@\,tweety$ results in $flies(tweety)$.

2.2 ASP

Answer Set Programming (ASP) is a form of logic programming based on the answer set semantics [7], where solutions to a given problem are represented in terms of selected models (answer sets) of the corresponding logic program [8, 9]. Rich literature exists on applications of ASP in many areas, including problem solving, configuration, information integration, security analysis, agent systems, semantic web, and planning (see among many [10–14] and the references therein).

In this logical framework, a problem can be encoded —by using a function-free logic language— as a set of properties and constraints which describe the (candidate) solutions. More specifically, an *ASP-program* is a collection of *rules* of the form

$$H \leftarrow L_1, \ldots, L_m, \mathit{not}\ L_{m+1}, \ldots, \mathit{not}\ L_{m+n}$$

where H is an atom $m \geq 0$, $n \geq 0$ and each L_i is an atom. The symbol *not* stands for negation-as-failure. Various extensions to the basic paradigm exist, that we do not consider here all of them as they are not essential in the present context. The left-hand side and the right-hand side of the clause are called *head* and *body*, respectively. A rule with empty head is a *constraint*. (The literals in the body of a constraint cannot be all true, otherwise they would imply falsity.)

The semantics of ASP is expressed in terms of *answer sets* (or equivalently *stable models*, [7]). Consider first the case of a ground ASP-program P which does not involve negation-as-failure (i.e., $n = 0$). In this case, a set of atoms X is said to be an answer set for P if it is the (unique) least model of P. Such a

definition is extended to any ground program P containing negation-as-failure by considering the *reduct* P^X (of P) w.r.t. a set of atoms X. P^X is defined as the set of rules of the form $H \leftarrow L_1, \ldots, L_m$ for all rules of P such that X does not contain any of the literals $L_{m+1}, \ldots, L_{m+n}$. Clearly, P^X does not involve negation-as-failure. The set X is an answer set for P if it is an answer set for P^X.

Once a problem is described as an ASP-program P, its solutions (if any) are represented by the answer sets of P. Unlike other semantics, a logic program may have several or no answer sets, because conclusions are included in an answer set only if they can be justified. The following program has no answer sets: $\{a \leftarrow not\, b,\ b \leftarrow not\, c,\ c \leftarrow not\, a\}$. The reason is that in every minimal model of this program there is a true atom that depends (in the program) on the negation of another true atom. Whenever a program has no answer sets, we will say that the program is *inconsistent*. Correspondingly, checking for consistency means checking for the existence of answer sets. For a survey of this and other semantics of logic programs with negation, the reader may refer to [15].

Let us consider the program P consisting of the three rules $\{r \leftarrow p,\ p \leftarrow not\, q,\ q \leftarrow not\, p\}$. Such a program has two answer sets: $\{p, r\}$ and $\{q\}$. If we add the rule (actually, a constraint) $\leftarrow q$ to P, then we rule-out the second of these answer sets, because it violates the new constraint.

To find the solutions of an ASP-program, an ASP-solver is used, where several solvers have became available [16]. The reader can see [10, 18], among others, for a presentation of ASP as a tool for declarative problem-solving.

2.3 CCG-Grammars

The Combinatorial Categorial Grammars (CCGs) [19, 20] have the aim of providing high expressive power while keeping automata-theoretic complexity to a minimum. CCG is a form of lexicalized grammar in which the application of syntactic rules is entirely conditioned on the syntactic type, or category, of their inputs. No rule is structure- or derivation-dependent. Categories identify constituents as either primitive categories or functions. Primitive categories, such as N (noun), NP (noun phrase), S (sentence), and so on, may be regarded as further distinguished by features, such as number, case, inflection, and the like. Functions (such as verbs) bear categories identifying the type of their result (such as VP, verb phrase) and that of their argument(s)/complements(s) (both may themselves be either functions or primitive categories). Function categories also define the order(s) in which the arguments must combine, and whether they must occur to the right or the left of the functor. Each syntactic category is associated with a logical form whose semantic type is entirely determined by the syntactic category. The slash '/' and '\' operators allow a category to combine by any combinatory rule.

In summary, a CCG grammar is composed of: a set of basic categories; a set of derived categories, each constructed from the basic categories; and some syntactical rules describing via the slash operators the concatenation and determining the category of the result of the concatenation. For instance, assume

that a CCG contains the following objects: $Tweety$ whose category is NP (noun phrase) and $flies$ whose category is (S\NP). The category of $flies$ being S\NP means that if an NP (a noun phrase) is concatenated to the left of $flies$ then we obtain a string of category S, i.e., a sentence.

Categories can be regarded as encoding the semantic type of their translation. This translation can be made explicit by associating a lambda-calculus logical form with the entire syntactic category. The parsing process will then generate a first-order lambda-calculus expression with quantifiers equivalent to the whole given sentence.

2.4 Automated Translation of Natural Language into ASP

The problem CCG do not cope with is that of *approximate* linguistic expressions that involve "fuzzy" assertions like, e.g., 'normally', 'most', etc. that do not have a direct correspondence in existentially/universally quantified sentences. Thus, an intermediate form is needed that is able to take this kind of sentences into account. This intermediate form should be such as to allow a translation/transposition into some executable formalism that allows the extracted knowledge to be reasoned about. Recent relevant work presented in [2] proposes the use of an intermediate λ-ASP-calculus representation, to be then translated into ASP. This calculus is an adaptation of λ-calculus to take the ASP rule format into account, so as to be able to represent fuzzy assertions and to translate them in a standard way into ASP.

The sample language discussed as a running example in [2], that they consider as a representative of a class of languages which are sufficient to represent an interesting set of sentences (including default statements and strong exceptions), is the following:

- Most birds fly.
- Penguins are birds.
- Penguins do swim.
- Penguins do not fly.
- Tweety is a penguin.
- Tim is a bird.

The first sentence is a *normative sentence* expressing a default: namely, it states that birds normally fly. The second sentence represents a subclass relationship, while third and fourth sentences represent different properties of the class of penguins. The last two sentences are statements about individuals.

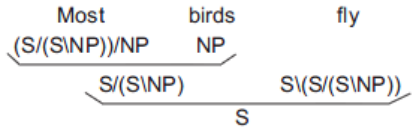
It is important to notice that none of previous approaches to automatic translation of natural language sentences is able to deal with default statements. The authors show how it is possible to automatically translate these sentences into ASP rules.

As first step, a CCG grammar for this sentences set is defined, L_{bird}. The CCG grammar is used because it gives information to "drive" the application of λ-expressions. However, as the goal is to obtain an ASP representation, the

notion of λ-representation is expanded to λ-ASP-expression which allows for construction of ASP rules. Then, the second step is the development of λ-ASP-expressions for words and categories in the language of interest. For example, for the L_{bird} grammar, the λ-ASP-expressions are (where variables denoting domain constants are indicated, as customary in ASP, in uppercase):

Word	Cat	$\lambda - ASP - expression$
fly	F1	x.fly(x)
	F2	x.fly(x) $\leftarrow$ x@X
most	M	$\lambda v.(v@X \leftarrow \lambda x.bird(x)@X,\ not\ \neg v@X)$
birds	-	$\lambda x.bird(x)$

Given this theoretical tool an example of translation is the follow. Given the sentence "Most birds fly", the CCG derivation tells how this sentence is constructed (where S=Sentence, NP=Noun phrase):



The word 'birds' is concatenated to the right of 'most', creating 'most birds'. This will be concatenated to the left of 'fly', creating a sentence whose category is S.

The λ-ASP-expressions for the categories of interest are:

M: $\lambda u \lambda v.(v@X \leftarrow u@X,\ not\ \neg v@X)$

B2: $\lambda x.bird(x)$

F1: $\lambda y.fly(y)$

The concatenation is driven by the CCG derivation, so the first step implies to concatenate 'birds' to 'most' and thus applying M to B2:

$(\lambda u \lambda v.(v@X \leftarrow u@X,\ not\ \neg v@X))@(\lambda x.bird(x))$ which gives

$\lambda v.(v@X \leftarrow \lambda x.bird(x)@X,\ not\ \neg v@X)$ which finally gives

$\lambda v.(v@X \leftarrow bird(X),\ not\ \neg v@X)$

The λ-ASP-expression for 'Most birds fly' is obtained by applying the above espression to F1, i.e.:

$(\lambda v.(v@X \leftarrow bird(X),\ not\ \neg v@X))@(\lambda y.fly(y))$ which gives

$\lambda y.fly(y)@X \leftarrow bird(X),\ not\ \neg \lambda y.fly(y)@X$ which finally gives

$fly(X) \leftarrow bird(X),\ not\ \neg fly(X).$

3 Enhanced Automated Translation of Natural Language into ASP

In this section, we propose an improved fully automated methodology for generating ASP rules from natural language sentences of the kind discussed above. In our proposal, we associate to grammar rules expressions defined in a meta-λ-ASP-calculus. These expressions are 'meta' in the sense that they are associated to general categories rather than, like in CCGs and in the work of [2], to specific instances. For example, we define an expression related to the syntactic category of names, to be instantiated to the specific occurrences. We are then able to automatically generate ASP rules. This alleviates the problem, mentioned in [2], that the construction of λ-expressions requires human engineering, and it is a first step towards their automatic generation.

We do not define our method on CCG grammars. In [2] it is observed that each CCG, whose only syntactical rules are the left- and right-concatenation rules, is equivalent to a context-free grammar (see, e.g., [21]) which can be used in syntactic checking. Parsing a sentence using a context-free grammar is often more efficient, but these grammars lack the directionality information that CCG has. For example, they will not indicate how to obtain the semantics of 'most birds' from the semantics of 'most' and 'birds'. I.e., whether to apply the λ-expression of 'most' to the λ-expression of 'birds' or vice-versa. To overcome this problem, we adopt SE-DCGs, a variant (that we have defined in previous work [5]) of the well-known and widely-used DCGs. 'Per se', the DCGs do not allow for the same expressivity of CCGs, as they are fully context-free. However, the SE-DCGs are equipped with built-in constraints that may occur in rules and are verified during the parsing process whenever they are encountered. This provides the context-sensitivity that we were lacking. The built-in constraints however, as they are evaluated on a background knowledge base, also allow for semantic analysis and disambiguation of sentences. This might be useful in the CCG context as well. I.e., our proposal is not necessarily bound to the DCG realm but may in practice be transposed elsewhere.

The resulting framework is fully logical, and as we had already implemented the SE-DCG's related tools we have implemented the new methodology on top of these tools so as to directly proceed with the experiments.

3.1 SE-DCGs

In [5] we have proposed an extension, called SE-DCGs (Semantic Enhanced DCGs), to the DCGs (Definite Clause Grammars) in order to overcome their purely syntactic nature and introduce forms of 'on-the-flight' semantic analysis/reasoning. The DCGs are a well-known useful feature of prolog systems. DCGs have been demonstrated to be convenient ways of representing grammatical relationships for various parsing applications. They can be used for natural language, for creating formal command and programming languages. Syntax of a DCG rule is quite intuitive: a sample rule is shown below, where non_term1 and non_term2 are non terminal symbols and $[term1]$ is a terminal symbol.

$non\_term1 \rightarrow non\_term2, [term1].$

On the left-hand side there is a non terminal symbol, while on the right-hand side there can be any sequence of terminal and non terminal symbols (for short 'terminals', indicated in square brackets).

The SE-DCGs can on the one hand improve the syntactic analysis and on the other hand allow one to elicit semantic information from sentences and periods. In terms of performance, the SE-DCGs are able to operate with high efficiency because of early search space pruning due to the prompt recognition of errors and inconsistencies. The extension of DCGs to SE-DCGs is very simple, and implies limited modifications to a DCG pre-processor. The proposed extension is the following. Non terminals may have (like DCG's) one or more logical variables as arguments, that will be instantiated during the parsing process to terminal symbols. However, the right-hand side of rules is enhanced by adding expressions in brackets that express constraints on these variables (that can be seen as embedded goals in the sense of [22]). Thus, a sample SE-DCG rule looks like:

$$non\_term1(X1, X2) \leftarrow non\_term2(X1), non\_term2(X2), [term1], \{constr(X1, X2, term1)\}.$$

where there can be as many arguments as needed, and each constraint (here indicated as $constr(X1, X2, term1)$) is a formula involving variables and terminals, as well as symbols occurring in the background knowledge base.

SE-DCG's have the potential of allowing one to check and extract semantic information from sentences and periods. Syntactic-semantic analysis is performed via prolog-like queries, e.g., for the sample sentence *Most birds fly*:

$$?- s(R, [most, birds, fly], []).$$

We obtain the parse tree below which correctly associates 'Most' to 'birds'.

$$R = element(subj(det('most'), noun('birds')), vp(vrb('fly'))))$$

In fact, starting from the *most* lexicon, the SE-DCG analysis determines is semantic role (class), in this case *det* [1].

The extracted knowledge will be stored in the knowledge base, and used by SE-DCG's embedded goals for semantical analysis and reasoning.

Syntactic and semantic enhancements are by no means exclusive and can coexist in the same SE-DCG. In fact, constraints are evaluated in the background knowledge base. The knowledge base will include a dictionary of known objects

[1] Even though traditional English grammar does not include determiners and calls most determiners adjectives there are, however, a number of key differences between determiners and adjectives. Modern English grammars include degree (or *quantity*) determiners like $many, much, few, little \ldots$ See Longman English Grammar by L. G. Alexander, R. A. Close for more details.

and ontological aspects to establish correspondences. In principle however, it may contain any form of either commonsense knowledge or scientific/specialized knowledge so as to possibly integrate reasoning about language with other forms of reasoning.

Constraints in SE-DCG rules allow the system to check, e.g., whether the terminals occurring in the sentence are known, and, if so, if they are used in a plausible way. For example, for a given sentence like 'airplanes flying in the blue sky', a constraint checking whether the subject can plausibly perform the action indicated by the verb, is performed by an SE-DCG's embedded goals like this:

$$plausibile\_subj(S, V), plausibile\_compl(V, C, Mode)$$

In the knowledge base, we might have rules such as:

$$plausibile\_subj(fly, V) :- flying\_object(V).$$
$$flying\_object(airplane)....$$

$$plausible\_compl(fly, P, place) :- fly\_where(P).$$
$$fly\_where(sky)....$$

The constraint ensures that the sentence is plausible, while the check would fail if for instance the sentence mentioning an airplane flying *in the blue sea* as *sea* is not included in the fly_where list. The reasoning process can also include repair actions, related to what to do either in case of failure or in case of uncertainty.

The contribution of the SE-DCGs is more evident as the semantics complexity of sentences to be translated increases. In fact, determining the correct semantic class membership for lexical elements in complex sentences often requires reasoning (for example for disambiguation, evaluation of plausibility, etc.). For instance, consider the sentence *John often flies AILines*: through plausibility checks it is possible to establish that the subject *John* is not a *flying_object*, and thus the subject is not able to fly by himself. Then, *plausible_compl* is useful for determining that *AILines* is not an adjective but rather it somehow indicates that John will fly through (by means of) AILines. Others examples and details can be found in [5]. SE-DCG's have been applied to a complex case study, Mnemosine, which is a prototype intelligent search engine taking as dataset the Italian Wikipedia pages.

3.2 Abstract λ-ASP-expressions

In our methodology, we associate grammar rules to expressions defined in a meta-λ-ASP-calculus. These expressions are 'meta' in the sense that they are associated to categories rather than to specific instances. This alleviates the problem of the construction of λ-expressions, and is a first step towards their automatic generation.

We define λ-ASP-expressions$_T$ (λ-ASP-Template-expressions) as a meta extension of λ-ASP-expressions. These expressions may contains lexical placeholders that are intended to be instantiated on the application context at hand. The instantiation of a λ-ASP-expression$_T$ expression produces a λ-ASP-expression.

We define β as the $\lambda - ASP - expression_T$ base.

For the running example, the λ-ASP-expression$_T$ Base (i.e., the knowledge base containing the related expressions) is:

Lexicon	SemClass	$\lambda - ASP - expression\ Template$
-	noun	$\lambda x. < noun > (x)$
-	verb	$\lambda y. < verb > (y)$
most	det	$\lambda u \lambda v.(v@X \leftarrow u@X,\ not\ \neg v@X)$

We have to suitably define the *instantiation* and *application* operations. The *instantiation*, denoted with the symbol @@, is the operation that will replace the lexical placeholder in a λ-ASP-expression$_T$ with the given parameter.
The syntax of *instantiation* is the follow:

$(\lambda - ASP - expression_T)@@(lexicon)$.

The result of instantiation is a λ-ASP-expression. For example, given the λ-ASP-expression$_T$:

$\lambda x. < noun > (x)$

The instantiation of *noun* with 'home', is done through:

$(\lambda x. < noun > (x))@@home$

and the resulting λ-ASP-expression is:

$\lambda x.home(x)$

An *application* operation is the transposition of the λ-calculus application to the λ-ASP-expression realm.

The translation of a given sentence into a corresponding λ-ASP-expression starts from the leaves of the parse tree and go backwards to the root symbol. The bottom-up visit of the parse tree drives the translation execution. For each terminal symbol an *instantiation* operation is performed, while each non-terminal implies an *application* operation to be performed.

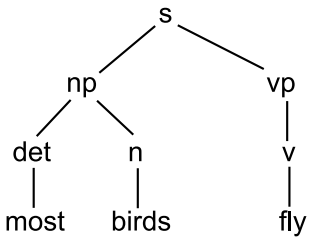
3.3 The Enhanced Methodology

The SE-DCGs play a key role, as the results of syntactic-semantics analysis are used to fully automatize the translation process. The semantic information is employed to find the correct λ-ASP-Template-expression from the λ-ASP-Template-expressions Base and thus instantiate or apply it automatically.

We illustrate the automatic translation process again with the help of the sentence *Most birds fly*. As seen above, the parse tree is represented as follows

$$R = (element(\\ subj(det('most'), noun('birds')), \\ vp(vrb('fly')))$$

and, graphically, looks like:



Starting the left-to-right bottom-up visit of the parse tree, we retrieve in the λ-ASP-Template-expression Base the class 'det' (semantic match) and the exact lexical match for the 'most' lexicon. Since *most* is a terminal symbol, according to previous definitions we have to perform an instantiation operation. The λ-ASP-Template-expression for *most* is $\lambda u \lambda v.(v@X \leftarrow u@X,\ not\ \neg v@X)$

By the instantiation $(\lambda u \lambda v.(v@X \leftarrow u@X,\ not\ \neg v@X))@@most$
we obtain the $\lambda - ASP - expression$: $\lambda u \lambda v.(v@X \leftarrow u@X,\ not\ \neg v@X)$

In this case, the instantiation operation returns the same λ-ASP-expression, because no placeholders needs to be instantiated. This happens when both semantic and syntactic match occurs. This is seldom the case in complex sentences, where *instantiation* will in general perform constrains checks and syntactic manipulation.

The next leave of the parse tree is the lexicon *birds*. It has the semantic role of "noun" in the context of the sentence. We find in the template base a match for all lexicons of the semantic class *noun*. As the *birds* lexicon is a terminal symbol, according to previous definitions we perform an instantiation. The appropriate λ-ASP-Template-expression is $\lambda x. < noun > (x)$,

The instantiation operation is performed with *birds* as parameter, obtaining: $(\lambda x. < noun > (x))@@birds$. Thus, the resulting $\lambda - ASP - expression$ is: $\lambda x.bird(x)$.

Going up the parse tree, we find non-terminal *np*. According to previous definitions, an *application* operations needs to be performed. In this case, semantic information drive the application of the λ-ASP-expression
$(\lambda u \lambda v.(v@X \leftarrow u@X,\ not\ \neg v@X))@(\lambda x.bird(x))$
and thus we get $\lambda v.(v@X \leftarrow \lambda x.bird(x)@X,\ not\ \neg v@X)$
which produces $\lambda v.(v@X \leftarrow bird(X),\ not\ \neg v@X)$

Now, we encounter the fly lexicon ($verb$), thus we look for a match concerning the *verb* semantic class, applicable to all lexicons of this class. We use this $\lambda - ASP - expression_T$ to *instantiate* the fly lexicon $(\lambda y. < verb > (y))@@fly$, and we get $\lambda y.fly(y)$.

For the *vp* class, the application is an identity, so we can skip to root symbol s, and thus perform the final application:

$(\lambda v.(v@X \leftarrow bird(X),\ not\ \neg v@X))@(\lambda y.fly(y))$
which returns
$\lambda y.fly(y)@X \leftarrow bird(X),\ not\ \neg\lambda y.fly(y)@X$
from which we get the final ASP expression:
$fly(X) \leftarrow bird(X),\ not\ \neg fly(X)$

4 Concluding remarks and future work

In this paper, we have introduced a fully automated process to translate natural language sentences into ASP theory, taking uncertain and defeasible knowledge into account. The method is an advancement over [2] in that it is based on an efficient semantically enhanced context-free class of grammars, and uses a more abstract intermediate representation that allows for automated translation via a visit of the parse tree. The proposed methodology has been implemented and experimented (the implementation is available from the authors).

In [2] it is observed that the class of sentences considered in their work so far lacks several constructs and in particular conjunctions (e.g. and, or, etc.), adverbs (e.g. quickly, slowly, etc.), and other auxiliary verbs (e.g. can, might, etc.) and that the problem in dealing with these constructs lies in making sure that each category can only be used in grammatically correct sentences and in ensuring that the semantical representation is proper. SE-DCGs with their 'on-line' semantic analysis can be of great use on these problems. For lack of space we cannot provide a proper explanation, we have already worked out sentences with conjunctions, like e.g. *Most birds fly and sing*.

The SE-DCG grammars that we have employed are a logic programming tool, and then we can say that the proposed methodology is fully logical, both in the definition and in the implementation. Experiments show that SE-DCGs seem to be able to retrieve enough semantic information to cope with complex sentences. However, we do not deny the superior expressive power of CCGs over DCGs. Therefore, our future work will include an attempt to merge the advantages of both.

References

1. Bos, J., Markert, K.: Recognising textual entailment with logical inference. In: HLT '05: Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing, Association for Computational Linguistics (2005) 628–635
2. Baral, C., Dzifcak, J., Son, T.C.: Using answer set programming and lambda calculus to characterize natural language sentences with normatives and exceptions. (2008) 818–823
3. Lassila, O., Hendler, J.: Embracing "web 3.0". IEEE Internet Computing **11**(3) (2007) 90–93
4. Berners-Lee, T., Hendler, J., Lassila, O.: The semantic web: A new form of web content that is meaningful to computers will unleash a revolution of new possibilities. Scientific American, issue of May, 17 (2001)

5. Costantini, S., Paolucci, A.: Semantically augmented DCG analysis for next-generation search engines. In: A. Formisano, ed., Online Proc. of CILC2008,, Italian Conference on Computational Logic. (2008) URL http://www.dipmat.unipg.it/CILC08/programma.html.
6. Moldovan, D.I., Harabagiu, S.M., Girju, R., Morarescu, P., Lacatusu, V.F., Novischi, A., Badulescu, A., Bolohan, O.: Lcc tools for question answering, TREC
7. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In Kowalski, R., Bowen, K., eds.: Proc. of the 5th Intl. Conference and Symposium on Logic Programming, The MIT Press (1988) 1070–1080
8. Lifschitz, V.: Answer set planning. In: Proc. of the 16th Intl. Conference on Logic Programming. (1999) 23–37
9. Marek, V.W., Truszczyński, M. In: Stable logic programming - an alternative logic programming paradigm. Springer (1999) 375–398
10. Baral, C.: Knowledge representation, reasoning and declarative problem solving. Cambridge University Press (2003)
11. Anger, C., Schaub, T., Truszczyński, M.: ASPARAGUS – the Dagstuhl Initiative. ALP Newsletter **17**(3) (2004) See `http://asparagus.cs.uni-potsdam.de`.
12. Leone, N.: Logic programming and nonmonotonic reasoning: From theory to systems and applications. In Baral, C., Brewka, G., Schlipf, J.S., eds.: Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007. (2007) 1
13. Truszczynski, M.: Logic programming for knowledge representation. In Dahl, V., Niemelä, I., eds.: Logic Programming, 23rd International Conference, ICLP 2007
14. Gelfond, M.: Answer sets. In: Handbook of Knowledge Representation, chapter 7. Elsevier (2007)
15. Apt, K.R., Bol, R.N.: Logic programming and negation: A survey. J. of Logic Programming **19/20** (1994) 9–72
16. : Web references of (some) ASP solvers
ASSAT: `http://assat.cs.ust.hk`;
Ccalc: `http://www.cs.utexas.edu/users/tag/ccalc`;
Clasp: `http://www.cs.uni-potsdam.de/clasp`;
Cmodels: `http://www.cs.utexas.edu/users/tag/cmodels`;
DeReS and aspps: `http://www.cs.uky.edu/ai/`;
DLV: `http://www.dbai.tuwien.ac.at/proj/dlv`;
Smodels: `http://www.tcs.hut.fi/Software/smodels`.
17. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and expressive power of logic programming. ACM Computing Surveys **33**(3) (2001) 374–425
18. Dovier, A., Formisano, A., Pontelli, E.: A comparison of CLP(FD) and ASP solutions to NP-complete problems. In Gabbrielli, M., Gupta, G., eds.: Logic Programming, 21st International Conference, ICLP 2005, Proceedings. Volume 3668 of LNCS., Springer (2005) 67–82
19. Steedman, M.J.: Gapping as constituent coordination. Linguistics and Philosophy **13**(2) (1990) 207–263
20. Steedman, M.J., Baldridge, J.: Combinatory categorial grammar. To appear in Robert Borsley and Kersti Borjars (eds.) Constraint-based approaches to grammar: alternatives to transformational syntax. Oxford: Blackwell, draft available on the web sites of the authors (2009)
21. Gamut, L.: Logic, Language, and Meaning. University of Chicago Press (1991)
22. Pereira, F.C.N., Shieber, S.M.: Prolog and natural-language analysis. Center for the Study of Language and Information, Stanford, CA, USA (1987)