Towards Translating Natural Language Sentences into ASP

Stefania Costantini and Alessio Paolucci

Dip. di Informatica, Università di L'Aquila, Coppito 67100, L'Aquila, Italy
stefania.costantini@univaq.it, alessio.paolucci@univaq.it

Abstract. We build upon recent work by Baral, Dzifcak 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 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 is 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 play 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 [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. In [1], [6] and [2], the authors try in particular to take into account sentences defining uncertain and defeasible knowledge. In this paper, we extend their approach by introducing a new more abstract intermediate representation to be instantiated on practical cases.

2 Background

Before entering into the details of our proposal, we need to introduce the necessary building blocks of the work of [2] and of our extensions. In [1], [6] and [2], the authors use CCG grammars to 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, ASP and CCG grammars, and then we illustrate the method of [2].

2.1 Preliminaries

Lambda Calculus λ-calculus is a formal system designed to investigate function definition, function application and recursion. 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, . . .):

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

Parentheses can be used for clarity. Lambda-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., $(\lambda x.x)y$ 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.

Reduction (also called β-reduction) is the only rule of computation. It concerns the replacement of a formal parameter by an actual one. It can only occur if a functional term has been applied to some other term. 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. β-reduction will be denoted by the connective @. Reduction is nothing other than the textual replacement of a formal parameter in the body of a function by the actual parameter supplied, for example $\lambda x.flies(x) \mathbin{@} tweety$ results in $flies(tweety)$. We can have λ expressions with various arguments, e.g., $\lambda x \lambda y.M$. In this case, we assume β-reduction to be possible on one variable at a time, and to be performed on the leftmost free variable. E.g., $\lambda x \lambda y.likes(x, y) \mathbin{@} mary$ results in $\lambda y.likes(mary, y)$ and $\lambda x.likes(mary, x) \mathbin{@} john$ results in $likes(mary, john)$. We use parentheses to indicate successive β-reductions, like in $((\lambda x \lambda y.likes(x, y) \mathbin{@} mary) \mathbin{@} john)$ that gives the same result as before.

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

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

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 since 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. We used Clasp solver [16]. The reader can see [10, 17], among others, for a presentation of ASP as a tool for declarative problem-solving.

CCG Combinatorial Categorial Grammars (CCGs) [18, 19] 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. A categorial grammar (CG) specifies a language by describing the combinatorial possibilities of its lexical items directly, without the mediation of phrase-structure rules like in traditional grammars. Consequently, two grammars in the same categorial grammar system differ only in the lexicon.

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, 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. In other words, the category S\NP of *flies* indicates that it is a verb, and that whenever it occurs in a natural language sentence its argument (namely the subject of the verb) is to be found at its left. As another example, sentence *mary likes joe* is of category (S\NP)/NP meaning that this time there are two arguments, one on the right and the other one on the left of the verb.

Categorial grammars have been widely used in linguistic research concerning semantics of natural language. In fact, lexical items are associated with semantic functions which correspond to the syntactic functions implicit in their categories. For instance, a phrase of category S/NP can be represented, from a semantic point of view, by a function from NP-type items to S-type items. By adopting λ-calculus, the sentence will be associated with a lambda expression of the form $\lambda x.\phi$, e.g., $\lambda x.flies(x)$. Similarly, for the category (S\NP)/NP we get $\lambda x.\lambda\phi$, e.g., $\lambda x \lambda y.likes(x,y)$.

2.2 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.

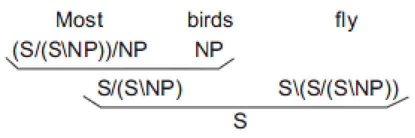
It is important to notice that none of previous approaches to automatic translation of natural language sentences is able to deal with default statements. The authors show how it is possible to automatically translate these sentences into ASP rules.

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	λ-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	-	$\lambda x.bird(x)$

Notice that the formula for 'most' is a schema to be instantiated to specific cases, while the other expressions are directly related to the example at hand.

Given this theoretical tool an example of translation is the following. 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 expression 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, i.e., involving determiners and thus uncertain knowledge. We remind the reader that the objective of producing an ASP representation of natural language sentences is that of adding the resulting ASP code to a knowledge base and then being able to reason and draw consequences from the knowledge extracted form the sentence. This on the one hand allows a system to enlarge its knowledge and on the other hand may improve the system capabilities: for instance, the system can provide the user with "intelligent" answers to her/his questions. Our proposal copes with the following aspects.

– **Meta-λ-ASP-expressions**
  In our proposal, we go farther in the direction of replacing domain-dependent λ-expressions with templates. We associate to grammar categories expressions which are 'meta' in the sense that they are associated to sets of sentences rather than, like in the work of [2], to specific instances. In particular, we allow meta-variables to denote function symbols in λ-expressions. We illustrate how to instantiate these meta-rules to the functional lexical elements thus obtaining specific expressions to be reduced w.r.t. their arguments. We will then be able to automatically generate ASP rules form this extended intermediate formalism. 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.

– **Managing several kinds of determiners, and conjunctions** The approach of [2] consider only the determiner 'most'. The reason of this lies in our opinion in the fact that other determiners, like 'some' or 'many' would produce basically the same expression, where what changes is the relative incidence of the set of exceptions. While 'most' admits few exceptions 'some' admits a lot of them, while 'many' is similar, though less committing, to 'most'. The representation of these subtle forms of commonsense connectives may profit from allowing meta-level statements to be represented in the background knowledge base that state the incidence of the exceptions. As we illustrate below, determiners 'some' and 'many' can be managed by including meta-level representations directly in the Meta-λ-ASP-expressions. A further improvement that we propose, in the direction of coping with more complex sentences, consists in dealing with conjunctions.

3.1 Abstract λ-ASP-expressions

In our methodology, we associate grammar rules to meta-λ-ASP-expressions. 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$_T$ (template λ-ASP-expressions) as a meta extension of λ-ASP-expressions. These expressions may contain lexical placeholders of the form $< nt >$, where $< nt >$ is a non-terminal of the given grammar, in place of function symbols. In the context of the meta-expressions, they play the role of meta-variables that are intended to be instantiated to functional representatives of these syntactic categories. I.e., basic template λ-ASP-expressions are of the form $\lambda x. < nt > (x)$ where $< nt >$ can be instantiated for instance to *flies*. The instantiation of a λ-ASP-expression$_T$ expression produces a λ-ASP-expression.

We also extend the templates of ASP rules to include meta-axioms to be evaluated in the background knowledge base in order to better represent plausible knowledge. We give below the example of determiners 'most', 'some' and 'many'.

Notice that we modify the definition of 'most' provided in [2]. In fact, we not only state that property v must hold for the set of objects u provided that given object is not an exception: we additionally state that for objects of kind u it is actually abnormal not to enjoy v. For determiner 'some' we take a complementary stance, stating that enjoying v is abnormal, e.g., in the sentence 'Some birds swim' (penguins, again!). Determiner 'many' is treated by stating that enjoying v is possible and also usual for objects of kind u. This and similar statements are able to cope with properties resulting from habits or preferences, like e.g. 'Many cats eat fish' or 'Many people like pasta'.

We define β as the $\lambda - ASP - expression_T$ Base, i.e., the set of meta λ-expression upon which the translation into ASP of given sentences is based. For the running example, the λ-ASP-expression$_T$ Base is:

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

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 following:

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

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 performed as follows:

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

and the resulting λ-ASP-expression is:

$$\lambda x.home(x)$$

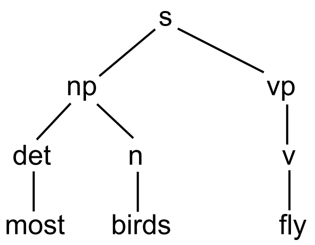
For the instantiation operation we adopt the same convention as for β-reduction, i.e., an instantiation refers to the leftmost metavariable, and parentheses have to be used in case of successive instantiations.

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.

3.2 The Enhanced Methodology

We illustrate the proposed methodology again with the help of the sentence *Most birds fly*. For the sake of simplicity, instead of the CCG parse tree we adopt in the example a more "traditional" parse tree, that looks as follows:



Starting a 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:

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

By the instantiation

$$(\lambda u \lambda v.(v@X \leftarrow u@X,\ not\ \neg v@X, abnormal(\neg v@X, u@X)))@@most$$

we obtain the $\lambda - ASP - expression$:

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

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* operation needs to be performed. In this case, semantic information drive the application of the λ-ASP-expression

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

and thus we get

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

which produces:

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

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:

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

from which we get the final ASP expression:

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

Below we illustrate the automatic translation process of another sentence, namely *Some robots walk*. This sentence make use of *some* determiner. The parse tree is trivial. We As first step we retrieve from the λ-ASP-Template Expression Base the expression that matches the class *det* (semantic match) and the *some* lexicon. *some* is a terminal symbol and thus we perform an instantiation operation. The λ-ASP-Template-expression for *some* is:

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

By the instantiation

$$(\lambda u \lambda v.(v@X \leftarrow u@X,\ not\ \neg v@X, abnormal(v@X, u@X)))@@some$$

we obtain the $\lambda - ASP - expression$:

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

The *robots* lexicon is the next leaf of the parse tree that we have to process. We find in the template base a match for all lexicons of the semantic class *noun* to which this lexicon belongs. According to definition we perform an instantiation from λ-ASP-Template-expression: $\lambda x. < noun > (x)$, obtaining:

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

Thus, the resulting $\lambda - ASP - expression$ is: $\lambda x.robot(x)$

Traversing the parse tree, we find the non-terminal symbol *np*. In this case, according to the semantic information the application of the λ-ASP-expression

$$(\lambda u \lambda v.(v@X \leftarrow u@X,\ not\ \neg v@X, abnormal(v@X, u@X)))@(\lambda x.robot(x))$$

produces:

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

When encountering the *walk* lexicon (*verb*), we find the match with the *verb* semantic class that is applicable to all lexicons of this class. We use this $\lambda - ASP - expression_T$ to *instantiate* the *walk* lexicon

$(\lambda y. < verb > (y))@@walk$, obtaining $\lambda y.walk(y)$.

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

$$(\lambda v.(v@X \leftarrow robot(X),\ not\ \neg v@X, abnormal(v@X, robot(X))))@(\lambda y.walk(y))$$

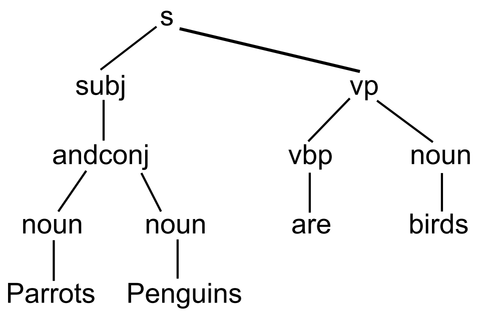
which produces

$$\lambda y.walk(y)@X \leftarrow robot(X),\ not\ \neg\lambda y.walk(y)@X, abnormal(\lambda y.walk(y)@X, robot(X))$$

Finally, we get the final ASP expression:

$$walk(X) \leftarrow robot(X),\ not\ \neg walk(X), abnormal(walk(X), robot(X))$$

We now illustrate how it is possible to translate sentences including conjunctions (like *and*). This is an important step that allows us to capture more complex sentences. The hardest problem in dealing with these constructs lies in the requirement to properly specify the correct category representing each word. [2] Assume to translate the following sentence: *Parrots and Penguins are birds*. The parse tree is:



The λ-ASP-expression$_T$ Base is extended to include more elements:

Lexicon	SemClass	$\lambda - ASP - expression\ Template$
-	noun	$\lambda x. < noun > (x)$
are	verb	$\lambda u \lambda v.v@X \leftarrow u@X$
-	verb	$\lambda y. < verb > (y)$
and	conj	$\lambda u \lambda v.u \wedge \lambda u \lambda v.v$
most	det	$\lambda u \lambda v.(v@X \leftarrow u@X,\ not\ \neg v@X, abnormal(\neg v@X, u@X))$
some	det	$\lambda u \lambda v.(v@X \leftarrow u@X,\ not\ \neg v@X, abnormal(v@X, u@X))$
many	det	$\lambda u \lambda v.(v@X \leftarrow u@X,\ not\ \neg v@X, possible(v@X, u@X), preferred(v@X, u@X))$

As a first step, we retrieve the $\lambda - ASP - Expression_T$ from the $\lambda - ASP - ExpressionTemplateBase$ for the semantic class *noun*, and *Parrots* lexicon.

According to our definition, we have to perform an instantiation, The λ-ASP-Template-expression for *noun* semantic class is $\lambda x. < noun > (x)$.

We use this $\lambda - ASP - expression_T$ to *instantiate* the *Parrots* lexicon

$(\lambda x. < noun > (x))@@parrot$,

and we get

$\lambda x.parrot(x)$.

Then we move to the Penguins lexicon. This lexicon was recognized as belonging to the noun semantic class by semantic analysis, so, similarly to previous step and according to the $\lambda - ASP - expression_T$ instantiation, we obtain

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

and we get

$\lambda x.penguin(x)$

We retrieve from the $\lambda - ASP - expression_T$ that the expression for the and conjunction is:

$\lambda u\lambda v.u \wedge \lambda u\lambda v.v$ shortened as $\lambda u\lambda v.u|v$

Note that due the different nature of translation process achieved through the parse tree returned by our enhanced methodology, the $\lambda - ASP - expression$ for the *and* conjunction is different from previous ones defined in literature [2]. In fact, we have to perform an instantiation with the two previous obtained sub-expressions as arguments.
From $\lambda u\lambda v.u|v$, the two arguments are

$\lambda x.parrot(x)$ and $\lambda x.penguin(x)$.

We obtain:

$(\lambda u\lambda v.u|v)@(\lambda x.parrot(x))$
$(\lambda v.(\lambda x.parrot(x))|v)$

and then

$(\lambda v.(\lambda x.parrot(x))|v)@(\lambda x.penguin(x))$

that reduces to:

$(\lambda x.parrot(x)|\lambda x.penguin(x))$

Moving to the right subtree, we find the lexican entry "are" that belongs to the semantic class *verb*, but as there is a lexical matching too we get an identity.

As there is a match in the Base, we have to perform an instantiation $(\lambda u\lambda v.v@X \leftarrow u@X)@@are$ thus obtaining:

$\lambda u\lambda v.v@X \leftarrow u@X$

Then:

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

That simplifies to:

$\lambda x.bird(x)$

Finally, we have to perform some $\lambda - calculus$ operations to obtain the final expression:

$(\lambda u \lambda v.v@X \leftarrow u@X)@(\lambda x.parrot(x)|\lambda x.penguin(x))$
$(\lambda v.v@X \leftarrow (\lambda x.parrot(x)|\lambda x.penguin(x))@X)$
$(\lambda v.v@X \leftarrow parrot(X)|penguin(X))$
$(\lambda v.v@X \leftarrow parrot(X)|penguin(X))@(\lambda x.bird(x))$
$(\lambda x.bird(x)@X \leftarrow parrot(X)|penguin(X))$
$bird(X) \leftarrow parrot(X)|penguin(X)$

To have an expression that is consistent with ASP definition we can divide the expression above into the two following rules:

$bird(X) \leftarrow parrot(X)$

and

$bird(X) \leftarrow penguin(X)$

4 Concluding remarks and future work

In this paper, we have introduced an advancement over [2] towards a fully for translating natural language sentences into ASP theories, taking uncertain and defeasible knowledge into account. In particular, we have proposed the adoption of meta-level axioms to be evaluated in a background knowledge base.

A main future direction is that of improving the present representation on the one hand by introducing more abstract templates and meta-axioms able to cope with functions with an arbitrary number of arguments and on the other hand by expressing more forms of plausible/uncertain knowledge, and dealing with the translation of complex sentences including other kinds of conjunctions and, in perspective, pronouns and adverbs.

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. : Clasp asp solver `http://www.cs.uni-potsdam.de/clasp`.
17. 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
18. Steedman, M.J.: Gapping as constituent coordination. Linguistics and Philosophy **13**(2) (1990) 207–263
19. 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)