FP7-ICT-2011.4.2
Contract no.: 288342
www.xlike.org

Deliverable D4.2.2

Final semantic triple graphs merging prototype

Editor:	Janez Starc, JSI
Author(s):	Janez Starc, JSI; Lluís Padró, UPC
Deliverable Nature:	Prototype (P)
Dissemination Level: (Confidentiality)	Public (PU)
Contractual Delivery Date:	M27
Actual Delivery Date:	M27
Suggested Readers:	All partners using the XLike Toolkit
Version:	1.0
Keywords:	knowledge extraction, micro-reading, knowledge base

Disclaimer

Full Project Title:	XLike– Cross-lingual Knowledge Extraction
Short Project Title:	XLike
Number and Title of Work package:	WP4 – Cross-lingual Semantic Integration
Document Title:	D4.2.2 Final semantic triple graphs merging prototype
Editor (Name, Affiliation)	Janez Starc, JSI
Work package Leader (Name, affiliation)	JSI
Estimation of PM spent on the deliverable:	6 PM

Executive Summary

The main goal of the XLike project is to extract knowledge from multi-lingual text documents and store it in a knowledge base. This deliverable presents an approach to micro-reading. The goal of micro-reading is to extract every possible fact in the input data.

In this approach a context-free grammar is induced and used to parse the textual input into semantic trees. Then, transformational functions are used to transform the semantic trees into semantic expressions that can be used for reasoning.

We conducted three experiments. In the first one the grammar is induced automatically, in second one semi-automatically. The third experiment is a combination of semantic role labelling and the developed approach.

Table of Contents

List of Figures

List of Tables

Abbreviations

CFG	Context-free grammar
Tf-Idf	Term frequency – Inverse document frequency
AM-TMP	Temporal role
AM-LOC	Locative role
AM-PNC	Purpose role

Definitions

Micro-reading	A knowledge extraction approach, where to goal is to extract every possible fact from the input data
Macro-reading	A knowledge extraction approach, where the goal is to extract a large number of facts from a large corpus, but not necessarily every fact
Context-free grammar	A set of recursive production rules used to generate patterns of strings
Production rule	A rewrite rule specifying a symbol substitution that can be recursively performed to generate new symbol sequences
Lexicon rule	Production rule, where the right-hand side consists of literals only
Transformational function	A function used to translate semantic trees into semantic expressions
Top-down parsing	A parsing strategy where one first looks at the highest level of the parse tree and works down the parse tree by using the rewriting rules of a formal grammar
CycL	The language of knowledge base Cyc
Bottom-up parser	Bottom-up parsing identifies and processes the text's lowest-level small details first, before its mid-level structures, and leaving the highest-level overall structure to last
Semantic role labelling	A task in natural language processing consisting of the detection of the semantic arguments associated with the predicate or verb of a sentence and their classification into their specific roles
Parallel Corpus	Parallel corpus consists of documents that are translated directly into different languages.
Comparable Corpus	Comparable corpus, unlike parallel corpora, contains no direct translations. Overall they may address the same topic and domain, but can differ significantly in length, detail and style.

1 Introduction

We have developed a knowledge extraction approach based on micro-reading [1], where the goal is to extract all possible facts from the textual input. On the other hand, we have developed a macro-reading approach in D4.2.1. The goal of macro-reading is to extract a large number of facts from a large corpus, but not necessarily every fact. The approach we have develop is based on patterns, as opposed to merging predicate-argument relations into semantic graphs, which is also part of task T4.2. In Section 4.3, we present a combination of both approaches.

2 Approach

We present an approach, where a context-free grammar is induced and used to parse the textual input into semantic trees. Then, transformational functions are used to transform the semantic trees into semantic expressions that can be used for reasoning.

2.1 Context-free grammar

We have developed an extended context-free grammar (CFG), where each production rule has one option and one transformational function assigned. This triple is named simply – ***rule***.
The grammar G is defined by the 6-tuple: G = (V, E, P, O, T, S), where

- *V* is the finite set of non-literals. Each non-terminal represents a semantic category. Therefore, the result of the parse is a semantic tree. We denote non-terminals with square brackets. For instance, *[Person], [Colour], [Organization].*
- *E* is the finite set of literals, which are all distinct tokens from all layers of the corpus. The corpus has several annotation layers, e.g. lexical, lemma, part-of-speech, named-entity, obtained by the tools from WP2 and WP3. However, the lexical layer is default in our experiments.
- *P* is a set of production rules that represents a relation from V -> (V u E)*, where the * represents the Kleene star operation. We divide the production rules on ***lexicon rules***, where the right side consists of literals only (V -> E*), and ***pattern rules***, where the right side contains at least one non-literal. Table 1 shows several examples of production rules. The first and the last examples are lexicon rules, while others are pattern rules.
- *T* is a set of transformational functions, which are presented in details in Section 2.4.
- *O* is a set of options, which are presented in details in Section 3.2.1.
- *S* is the starting non-terminal symbol. Since non-terminals represent semantic categories, the starting symbol is chosen based on the semantic category of the input examples. If the input examples are sentences, then the appropriate category may be *[Relation]*. While if the input examples are noun phrases, the starting symbol may be a more specific category, like *[Furniture type]* or *[Job Title].*

	Production Rule	Option	Transformation rule
1)	[Person] ::= napoleon	none	Const(Napoleon)
2)	[LifeRole] ::= a [LifeRole]	none	Blank()
3)	[LifeRole] ::= former [LifeRole]	none	Fun(FormerFn)
4)	[Relation] ::= [Person] is [LifeRole]	none	Pred(lifeRole)
5)	[Location] ::= [Country]	full_parse	SubType()
6)	[Person] ::= it	negative	N/A

Table 1- Examples of rules

2.2 Semantic trees

The top-down parser, presented in Section 3.2, constructs a *semantic tree* from one textual example, e.g. sentence or phrase, using the context-free grammar. An example of a semantic tree is presented on Figure 1. Since the context-free grammar is phrase structure grammar, each node has its own semantic category

and phrase. Each node was developed by a single rule. When the semantic tree is transformed into semantic expression, each node is evaluated by a transformational function from that rule.

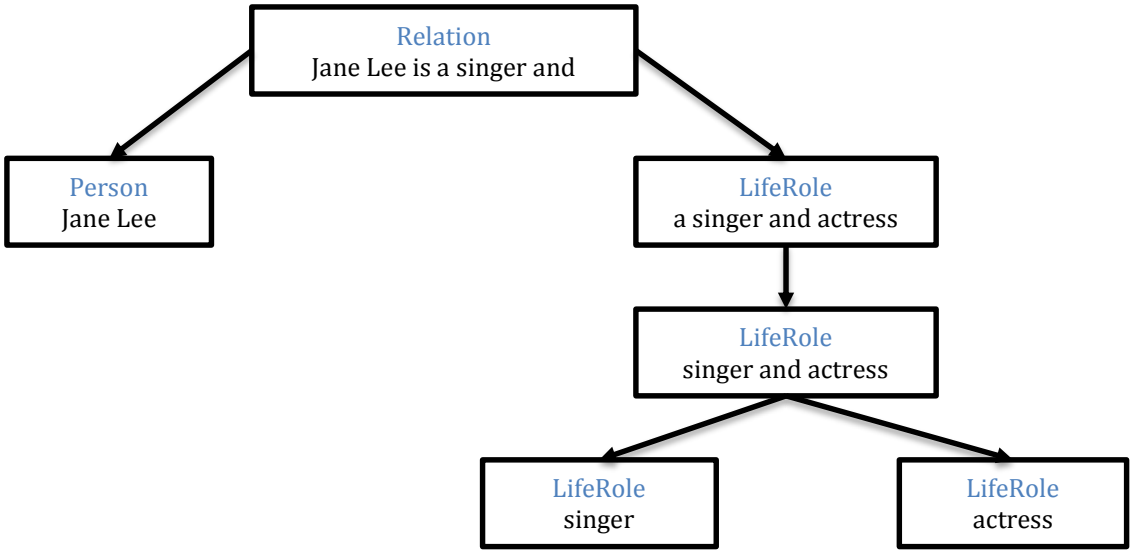

Figure 1- Example of a semantic tree

2.3 Semantic expressions

The final output of the parsing and transformation of one textual example is a semantic expression. The knowledge representation can be very arbitrary, and it is defined by transformational functions. The formal system that we use is a first-order logic. The language of semantic expression is very similar to CycL, the language of Cyc knowledge base. This is an example of semantic expression transformed from the semantic tree on Figure 1:

```
(and
        (lifeRole JaneLee (CollectionIntersectionFn Singer Actor))
        (isa JaneLee Person)
        (isa Singer LifeRole)
        (isa Actor LifeRole)
)
```

2.4 Transformational functions

When the *transformational function* is applied on a semantic sub-tree, it produces a semantic expression. To transform the whole semantic tree into a semantic expression, the transformational functions are applied recursively. On each node the transformational function uses the semantic expressions of sub-nodes to construct a new semantic expression. Each transformational function consist of a functor, which defines the type of transformation, and arguments, which are usually concepts from the knowledge base that are combined to construct the expression. From the technical point of view, the transformation function is a reflection method. When it is invoked, its code is being executed. If there is more than one relation in the final expression, they are connected with logical *and* operator, to form a valid logical expression. In our work we have defined several functors:

- The **constant** functor *Const* is used on the leaf nodes of the semantic tree. It has only one argument, which is the result of the function when applied. Additionally, an isa-relation is added to the semantic expression. For, example if the semantic category of the node is *[Person]* and the transformational function is *Const(Napoleon)*, than *(isa Napoleon Person)* is added to the semantic expression.
- The **blank** functor *Blank* is used on the nodes which have no semantic meaning. For example, to deal with some tokens, like *a*, *an*, *the*, period, which have no direct semantic meaning. This functor has

no arguments, and can be used only on nodes that have only one sub-node. The result of the transformational function is the result of its sub-node.

- The **function** functor *Fun* is used to construct semantic expressions that represent entities, which are constructed using semantic functions. For example, (PresidentFn France) represents the collection of presidents of France, (ColorFn Black Car) represents the collection of black cars, and (CollectionIntersectionFn Singer Actor), represents the collection of people that are both singers and actors. Many more semantic functions can be found in Cyc knowledge base. The semantic function is the only argument of function functor. When applied, the arguments of the semantic function are evaluated sub-nodes of the current node.
- The **predicate** functor *Pred* is evaluated similarly then the function functor. However, the result of the predicate functor is not an entity, which can be nested into other expressions, but an independent relation.
- The **sub-type** functor SubType is used with production rules, which have no literals, and only one non-literal, like *[Location] ::= [Country]*. The functor is used to express a sub-type relation, like (subtype Location Country).

3 Grammar induction

This section describes the approach to induction of our grammar described in Section 2.1. It is depicted on Figure 3. First, seed lexicon rules, which are obtained by the process described in Section 3.1, are added to the grammar. This is followed by an iterative procedure. On each iteration one production rule is constructed. The input at the start is a set of textual input examples. Each input example is a tokenized and annotated fraction of the corpus, e.g. sentence or noun phrase. The tokenization and annotation tools from WP2 and WP3 are used. First each input example is parsed using the top-down parser described in Section 3.2. Some sub-phrases of the input examples cannot be parsed yet. These sub-phrase are then generalized using the bottom-up parsing algorithm, described in Section 3.3. Considering the list of patterns obtained by generalization, a new rule is built, either automatically or semi-automatically, and added to the set of existing production rules (see Section 3.4). The iterative procedure is repeated until a certain stopping criteria is met.

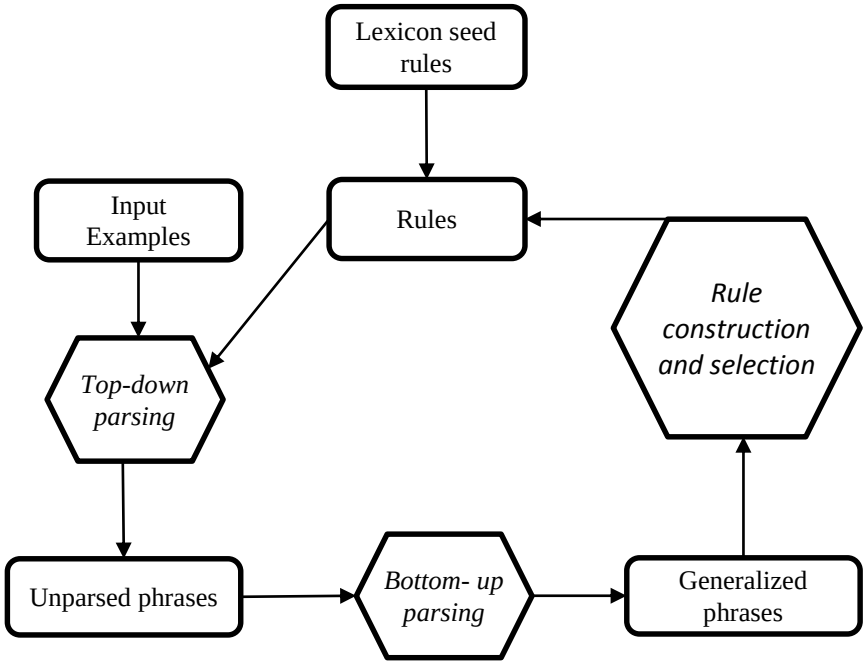

Figure 2 - Diagram of the approach

3.1 Seed lexicon rules

Since the whole process is based on bootstrapping of production rules, an initial set of lexicon rules is required. These can be obtained from a knowledge base. The Semantic Web offers a lot of sources like, Freebase, DBpedia, OpenCyc, from which the lexicon rules can be obtained. In this knowledge bases, we search for sub-graphs, like the one on

Figure 3, and transform them into lexicon rules. The names for predicates and constants on Figure 3 may differ across knowledge bases. However, most knowledge bases contain such relations. Lexicon rules always have a constant functor in the transformational function. The object (*Object)* defines the sole argument of the transformation function. The object is connected to its type (*Type)* – with *typeOf* predicate. The type defines the right-hand side non-terminal of the production rule. The object is also connected with its name (*Literal)* with *label* predicate. The name defines the right-hand side of the production rule. Using the sub-class relation (*subClassOf*), it is also possible to infer other types of the object. An object can also have multiple types and names (aliases). In this case every possible type-object-name fact link forms one production rule. This may lead to a vast number of lexicon rules that may be never used. To eliminate such

rules we employ policies for making the grammar *proper.* The grammar is proper if it does not contain any inaccessible non-terminals and all literals are also found in the input examples.

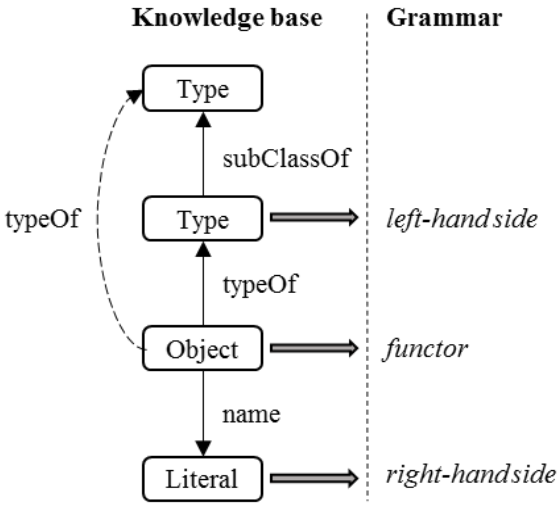

Figure 3 - Seed rule construction

3.2 Top-down parsing

In the parsing step, the *top-down parser* attempts to parse input examples one by one. In this process, the parser attempts to develop each node, and works down the parsing tree in a breadth-first manner. Developing a node means that the parser attempts to find a production rule that matches on one hand the non-terminal of the node, and on the other hand the lexical input of that node by employing a pattern matching algorithm. If a node cannot be developed then the whole example is unparsed. However, the parsing continues until every node has been attempted to develop. The nodes that could not be developed are used for production rule suggestions, which we describe in Section 3.4. At this point, we just mention that the lexical input and the non-literal (category) of each undeveloped node is stored for that purpose. There are two kinds of ambiguities that might occur in the process of parsing. The first one occurs when the same production rule develops a node in more than one way, while the second one occurs, when two rules develop the same node. To resolve ambiguities we use the greedy principle.

3.2.1 Options

Each rule can have one of three options that are considered by the top-down parser:

- Option ***none*** is a default option. If top-down parser encounters this option, it works normally as explained in Section 3.2.
- If the top-down parse encounters option ***full_parse*****,** than unparsed phrases, either at the current node or at any of its sub-nodes, are not added to the list of unparsed phrases. This options should be used on production rules that are frequently matched, but are rarely fully parsed. This option prevents misleading phrases to be added to list of unparsed phrases. Usually, these production rules have very short right-hand sides. For example, consider production rule *[Attribute] ::= born [Date]* and phrase “born 1950 in Rome”. If the option of this rule was *none*, then 1950 in Rome would be added to the *[Attribute]* unparsed list. This would bring a lot of noise to the rule suggestions.
- Option ***negative*** is used to catch wrong parses. We will name production rules with negative option negative rules. If top–down parser matches a negative rule than the rule that was matched in the parent node is now considered unmatched and the unparsed phrases of the negative rules’ sister nodes are not added to the unparsed lists. The top-down parser continues at the parent node, trying to match remaining production rules. An example of a negative rule is *[Person] ::= it*.

3.3 Bottom-up parsing

At each step of the iteration, the top-down parser (Section 3.2) parses the remaining input examples. If an example is successfully parsed, then it is withdrawn from the set of input examples. Otherwise, there was at least one undeveloped node. The top-down parser stores <category, input sub-phrase> pairs for all undeveloped nodes. Each sub-phrase is generalized by the *bottom-up parsing algorithm*. The generalized phrases will present the right-hand side of new production rules. The generalized phrase consist of non-literals (categories), and literals that could not be generalized. An example of bottom-up parse is presented on Figure 4. At the beginning the phrase consist of literals only. On each step, the bottom-up parser searches for an existing production rule that has a right-hand side that matches a part of a phrase. If such rule is found, the left-hand side (non-literal) replaces the matching sub-string. Therefore, the bottom-up parser is basically a string search and replace algorithm, which stops when no production rules can be applied anymore. The parser uses the Aho-Chorasick string matching algorithm [2], which is efficient, when dealing with multiple search patterns. The parser works from left to right of the input phrase. If more than one production rule applies at a certain position, then the algorithm applies the longer one. Therefore, the final generalized phrases tend to be shorter, which makes new production rules less complex. Due to the greediness of the bottom-up parser, the result is only one generalized phrase, although, ambiguous parses would be possible. However, this would slow down the algorithm, and also the remaining steps of the iteration.

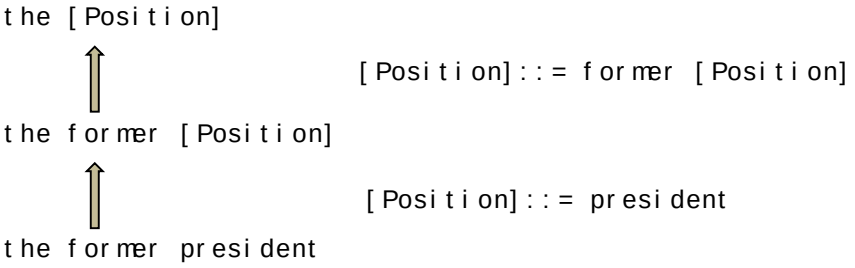

Figure 4 Example of bottom-up parsing

3.4 Rule construction and selection

At this point, there exists a set of pairs [categories, generalized phrase]. For each category generalized phrases are merged and counted. Consequently, each category has a list of generalized phrases sorted by frequency. To form a new production rule, the category of the list is taken as a left-hand side and one generalized phrase from the list as the right-hand side. In the next step, one production rule is selected from all lists. Then an option and transformational function are added to the production rule to form a new rule. In Section 4.1, we describe a scenario, where rules are created automatically. In Section 4.2 we describe a scenario, where rules are constructed semi-automatically. The newly constructed rule is added to the bottom of the grammars' rule list. Consequently, rules that are selected earlier, have higher priority in the grammar. The order of generalized phrased in a particular list may change over iterations. This is the reason, why only one rule is constructed on each iteration. The tendency is to have, as generalized rules as possible. Adding rules one by one may cause one more general rule, which started low in the list, to climb up this list and overtake more specific rules.

4 Experiments

4.1 Automatic grammar induction

We have conducted an experiment, where there was no human interference in the grammar induction process. The goal of this experiment was to extract roles of people in organization. First we applied two simple patterns on the corpus to obtain input examples:

- [person] , ?ROLE in [organization]
- [person], ?ROLE in the [organization]

Literals [person] and [organization] are tokens from the named-entity layer. All that tokens that replace the *?ROLE* are the input examples for the grammar induction. Since we are looking for roles, we defined the [Role] category, and made it the starting symbol S of the grammar. We used Freebase to create the seed lexicon rules. We extended the Freebase ontology, in a way that the following freebase types were subtype of the role category: *Profession, Organization committee title, Project role, Board Member Title, Religious Leadership Title, Legislative committee title, Government office category, Leadership Role, Academic post title, and Editor title*. Then, we applied the procedure from Section 3.1 to obtain seed lexicon rules, and retained only the one that had [Role] as the left-hand side non-literal. Therefore, the [Role] is the only category, and there was just one rule suggestion list. A new rule was made from the top candidate from this list, like it is described in Section 3.4. All new rules had option *none*, and transformational function with either the function functor if production rule was a pattern rule, or constant functor if the production rule was a lexicon rule. The sole argument of the function was automatically created from the non-literals of the production rule. For instance, from *[Role] ::= former [Role]*, the argument Fun(FormerFn) was created.

4.2 Semi-automatic grammar induction

In this experiment, the input data were first sentences of Wikipedia pages representing people. For example:

Nicanor Parra Sandoval (born 5 September 1914) is a Chilean poet, mathematician, and physicist.

Like all Wikipedia pages, these sentences have links to other Wikipedia pages. We use Freebase ontology to get the categories (types) of this pages that are linked. These links are used to construct lexicon rules. The process is analogous to the process of obtaining seed lexicon rule (Section 3.1). The category represent the left-side of a production rules. The text, which is linked, represents the right-side of a production rule. These lexicon rules can be applied only on the part of the text, they were created from. Therefore, these rules represent just another layer in the textual representation, besides lexical, lemma, part-of-speech, etc.
Since entities have several categories, there is usually more than one lexicon rule created for one link. Before bottom-up parsing, all rules representing one link are joined into one super rule, where the left-hand side is a super category - *[Entity].* When suggesting rules, lots of generalized phrases were merged into a phrase containing a super-category. In the process, the super-category is replaced by a category that best represents that position in the production rule. We used a metric similar to tf-idf (term frequency-inverse document frequency) to determine the best category. In our case, the *term frequency* is the fraction of times a category appears in all sub-rules, and *document frequency* is the number of links, whose entities are of that category. The grammar was crated semi-automatically. A human annotator controlled the process of constructing rules. His main interventions in this process were:

- Adding option and transformational function to the production rule
- Changing order of rules
- Adding rules that were not suggested when needed. For example, rule 4) from Table 1 is very important to bootstrap the process. Some essential rules were also created from scratch. However, these can be also used in other domains. For instance, rule that parses conjunction (*[Entity] and [Entity]*).
- In some rules, the tf-idf metric suggestion for categories was not optimal. Therefore, some non-terminals were slightly changed.

4.2.1 Evaluation

The input for grammar induction was 7981 first sentences of person Wikipedia pages. There were 87 rules constructed. The evaluation criteria for this experiment was coverage (recall) – how many sentences get completely extracted. The other possible criteria would be precision. However, an annotator constructs the grammar for his needs – knowledge base. He is the only one who can judge the correctness of the results. Therefore, we believe that measuring the precision is subjective. Furthermore, we argue here that there is no big need for a separate training set, since we are optimizing the coverage of the input data.
The input set was parsed with the constructed grammar. There were 918 fully parsed sentences, which is about 11.5% of the input data. The parsing took about six seconds on a single thread. Examples of rules and example semantic extractions are presented in Annex A.

4.3 Grammar induction based on semantic role labelling

This experiment was similar to previous one. Instead of first sentences of person Wikipedia pages, we took first paragraphs. Furthermore, instead of parsing whole sentences, we used semantic role labelling to limit the parsing to sub-phrase of the sentence, and to obtain additional semantic information. Figure 5 shows an example of a sentence, which has one predicate *play*, which has four roles – A0, A1, AM-TMP, AM-LOC. A node in a syntactic tree is mapped to each role. The input for parsing is the phrase that is represented by the sub-tree of that node.

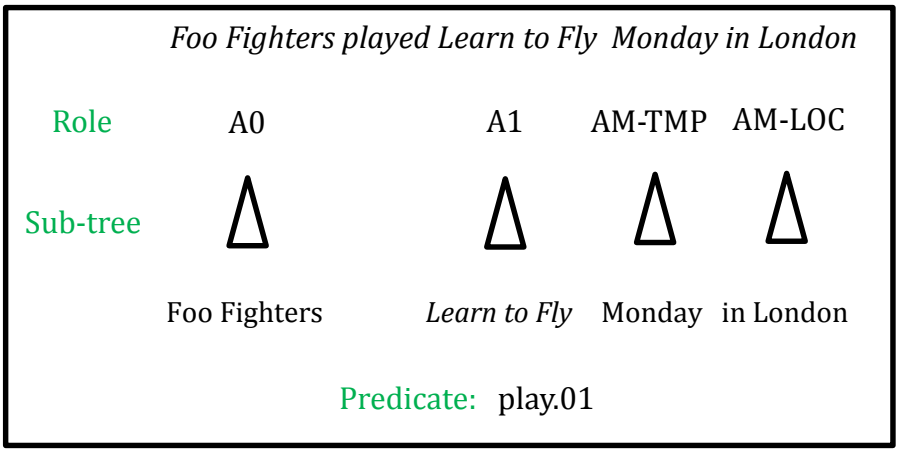

Figure 5- Semantic role labelling example

But before parsing, a concept of event has to be assigned to the predicate. Furthermore, each role has to be assigned a semantic type, which will be the starting symbol *S* for the parsing, and a predicate that connects the role to the predicate. Figure 6 shows such mappings for the roles for the example on Figure 5. The list of all roles can be found in [3], and they are divided in four types. We will consider the two more important types: *Numbered roles arguments (A0, A1, A2, etc.)* and *adjuncts (AM-TMP, AM-LOC, AM-PNC, etc.)* The actual semantic role for adjuncts is the same for every predicate. For numbered arguments, the actual semantic role differs across different predicates. Therefore, the categories and semantic predicates can be defined for every predicate separately, or we can define some general categories and semantic predicates that has less semantic expressiveness. To be able to parse more text with semantically richer expressions, it is useful to create mappings for predicates and roles that are more frequent. A human annotator that defines such mappings may look into to frequency tables to select the right predicates.

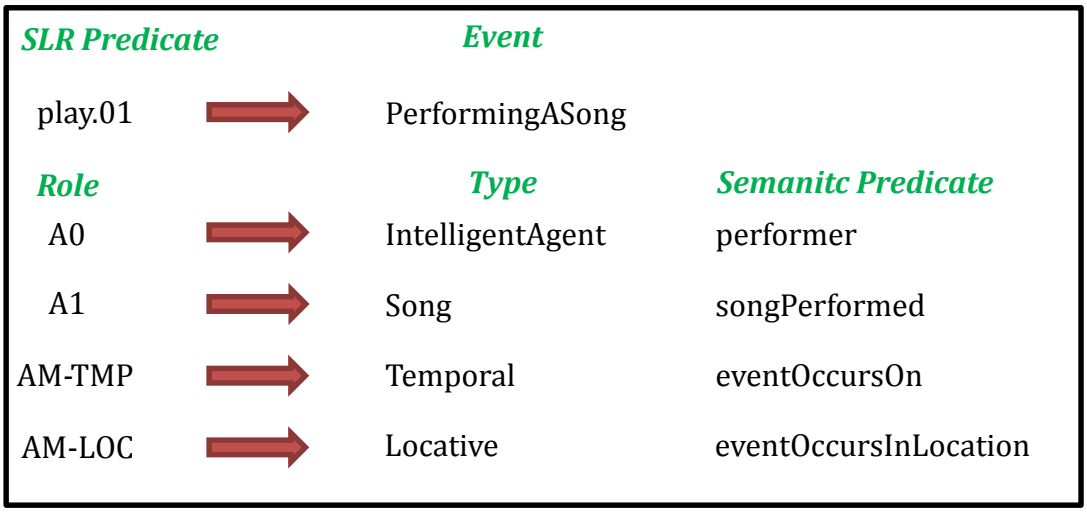

Figure 6 - Semantic role labelling mappings

Each role is parsed separately using the approach from Section 2 to obtain a semantic expression. Then all expressions are combined to construct the final semantic expression. The final semantic expression for our running example is:

```
(thereExists ?EVENT
        (isa ?EVENT PerformingASong)
        (performer ?EVENT FooFighters)
        (songPerformed ?EVENT LearnToFly-Song)
        (eventOccursOn ?EVENT Monday-DayOfWeek)
        (eventOccursInLocation ?EVENT London)
```

5 Conclusion

We presented an approach to macro-reading using context-free grammar. We believe that context-free grammar is very suitable for this approach, because it is expressive enough to express basic semantic phenomena, and fast enough to parse a few thousand sentences in a few seconds. To make the extracted knowledge capable of reasoning we use the transformational functions, which connect semantic trees with the vocabulary (ontology) of the knowledge base. We have not spotted a knowledge base that contains vocabulary, which is capable of expressing all kinds of facts that are encoded in sentences, like the ones from Wikipedia. Therefore, the induction of grammar also serves as the definition of vocabulary of the knowledge base.

All our experiments were conducted only on English texts. However, for other languages the approach would not change much. The pre-processing phase to tokenize sentences and obtaining various syntactic and semantic layers is available from WP2 and WP3 for several languages. There also exist knowledge bases, like Freebase, which can be exploited for creating seed lexicon rules. In future, it would be interesting to make grammars for several languages on parallel or comparable corpora (for instance, Wikipedia page representing the same topic in several languages), and observe the obtained knowledge that would be expressed in the same semantic language. Since, the grammar also works in the opposite direction, it could be used to generate language. Then, facts that are missing in the document of one language could be supplemented.

Besides, the extension to other languages, we would like to make the process even more automatic in the future. There is a possibility of using self-supervision to evaluate the correctness of an automatically selected rule. Furthermore, the ambiguities in the process could be resolved by making semantic trees stochastic, instead of using greedy methods.

References

[1] Mitchell, T. M., Betteridge, J., Carlson, A., Hruschka, E., & Wang, R., "Populating the semantic web by macro-reading internet text.," in *The Semantic Web-ISWC 2009*, Berlin, 2009.

[2] Aho, A. V., & Corasick, M. J. , "Efficient string matching: an aid to bibliographic search.," *Communications of the ACM,* vol. 18, no. 6, pp. 333-340, 1975.

[3] Carreras, X., & Màrquez, L., "Introduction to the CoNLL-2005 shared task: Semantic role labeling," in *Proceedings of the Ninth Conference on Computational Natural Language Learning*, 2005.

Annex A Examples of rules and semantic extractions from Section 4.2

Rules:

```
<Relation> ::= <Relation> .                    || none     || Blank()
<Relation> ::= <Person> .                      || negative || Blank()
<Relation> ::= <Organization> .                || negative || Blank()
<Relation> ::= <Person> is <LifeRole>          || none     || Pred(lifeRole)
<Relation> ::= <Person> was <LifeRole>         || none     || Pred(lifeRole)

<Person> ::= <Person> ( )       || none      || Blank
<Person> ::= <Person> ( <Attributes> )     || none        || Pred(attributes)
<Person> ::= <Person> ( ) ( <Attributes> ) || none        || Pred(attributes)
<Person> ::= <Person> , <OrderOfChivalry> || full_parse || Pred(orderOfChivalry)

<Attributes> ::= born <Date> in <Location> || none || Fun(BirthDatePlaceFn)
<Attributes> ::= born <Person> ; <Date>    || none     || Fun(AliasBirthDateFn)
<Attributes> ::= born <Person> on <Date>   || none     || Fun(AliasBirthDateFn)
<Attributes> ::= born <Date> - <Date>      || none     || Fun(BirthDeathDateFn)
<Attributes> ::= born <Date> - <Date>      || none     || Fun(BirthDeathDateFn)
<Attributes> ::= born née <Person> ; born <Date> || none ||Fun(NeeNameBirthFn)
<Attributes> ::= born <Date> <Date>        || none     || Fun(BirthDeathDateFn)
<Attributes> ::= born <Date>               || full_parse || Fun(BirthDateFn)
<Attributes> ::= born <Person> , <Date>    || full_parse || Fun(AliasBirthDateFn)

<LifeRole> ::= <Title> from <Date> to <Date> || none|| Fun(TitleStartEndDateFn)
<LifeRole> ::= <Title> from <Date> until <Date> || none ||
Fun(TitleStartEndDateFn)
<LifeRole> ::= [misc] <LifeRole> || none || Fun(CollectionIntersectionFn)
<LifeRole> ::= [Location] n <LifeRole> || none  || Fun(NationalityFn)
<LifeRole> ::= [Location] <LifeRole>   || none || Fun(NationalityFn)
<LifeRole> ::= [Ethnicity] <LifeRole>  || none || Fun(EthnicityFn)
<LifeRole> ::= former <LifeRole>       || none || Fun(FormerFn)
<LifeRole> ::= retired <LifeRole>      || none || Fun(RetiredFn)
<LifeRole> ::= <LifeRole> , who <PersonRelation> || none|| Fun(PersonRelationFn)
<LifeRole> ::= <LifeRole> who <PersonRelation>  || none|| Fun(PersonRelationFn)
<LifeRole> ::= founder of <Organization>  || none || Fun(FounderOfFn)
<LifeRole> ::= son of <Person>      || none || Fun(SonFn)
<LifeRole> ::= professor of <FieldOfStudy> || none || Fun(ProfessorFn)
<LifeRole> ::= <Discipline> ist || none || Fun(AwardDisciplineIstFn)
<LifeRole> ::= <LifeRole> from <Location> || none || Fun(PersonRelationFn)
<LifeRole> ::= member of the <Group> || none || Fun(MemberFn)
<LifeRole> ::= film <LifeRole> || none || Fun(FilmRoleFn)
<LifeRole> ::= <Genre> author || none || Fun(GenreAuthorFn)
<LifeRole> ::= businessman || none|| Const(Businessman)

<LifeRole> ::= [Profession]          || none     || SubType()
<LifeRole> ::= [Job_title]           || none     || SubType()
<LifeRole> ::= [Field_Of_Study]      || none     || SubType()
<FieldOfStudy> ::= [Field_Of_Study]  || none     || SubType()
<Discipline> ::= [Award_discipline]  || none     || SubType()
<Group> ::= <Organization>           || none     || SubType()
<Group> ::= [Family]                 || none     || SubType()
<Genre> ::= [Literary_Genre]         || none     || SubType()
<Title> ::= [Religious_Leadership_Title] || none || SubType()
<Title> ::= [Government_Office_or_Title] || none || SubType()
```

Extractions:

John Joseph Travolta (born February 18, 1954) is an American actor, dancer, and singer.

```
(isa (lifeRole JohnTravolta (CollectionIntersectionFn American (IntersectionFn Actor Dancer Singer))) Relation)
(isa JohnTravolta Person)
(attributes JohnTravolta (BirthDateFn February_18_,_1954))
(isa (BirthDateFn February_18_,_1954) Attributes)
(isa February_18_,_1954 Date)
(isa (CollectionIntersectionFn American (IntersectionFn Actor Dancer Singer)) LifeRole)
(isa American LifeRole)
(isa (IntersectionFn Actor Dancer Singer) LifeRole)
(isa Actor LifeRole)
(isa Dancer LifeRole)
(isa Singer LifeRole)
(lifeRole JohnTravolta (CollectionIntersectionFn American (IntersectionFn Actor Dancer Singer)))
```

Peter Greenaway, CBE (born 5 April 1942) is a British film director.

```
(isa (lifeRole PeterGreenaway (CollectionIntersectionFn British (FilmRoleFn Director))) Relation)
(isa PeterGreenaway Person)
(attributes PeterGreenaway (BirthDateFn 5_April_1942))
(isa (BirthDateFn 5_April_1942) Attributes)
(isa 5_April_1942 Date)
(orderOfChivalry PeterGreenaway OrderOfTheBritishEmpire)
(isa OrderOfTheBritishEmpire OrderOfChivalry)
(isa (CollectionIntersectionFn British (FilmRoleFn Director)) LifeRole)
(isa British LifeRole)
(isa (FilmRoleFn Director) LifeRole)
(isa Director LifeRole)
(lifeRole PeterGreenaway (CollectionIntersectionFn British (FilmRoleFn Director)))
```

Anita O' Day (October 18, 1919 November 23, 2006) was an American jazz singer.

```
(isa (lifeRole AnitaO'Day (CollectionIntersectionFn American (SingerOfGenreFn Jazz))) Relation)
(isa AnitaO'Day Person)
(attributes AnitaO'Day (BirthDeathDateFn October_18_,_1919 November_23_,_2006))
(isa (BirthDeathDateFn October_18_,_1919 November_23_,_2006) Attributes)
(isa October_18_,_1919 Date)
(isa November_23_,_2006 Date)
(isa (CollectionIntersectionFn American (SingerOfGenreFn Jazz)) LifeRole)
(isa American LifeRole)
(isa (SingerOfGenreFn Jazz) LifeRole)
(isa Jazz LifeRole)
(lifeRole AnitaO'Day (CollectionIntersectionFn American (SingerOfGenreFn Jazz)))
```