Lexical Class Semantic Analysis

Ivan Habernal and Miloslav Konopík

***Abstract*— Semantic analysis of *lexical classes* is a fundamental step of semantic analysis based on stochastic semantic parsing. The lexical class is a single word or a word group with specific semantic information such as dates, times, cities, etc. Having obtained a set of lexical classes, a semantic parse tree can be built upon it. This tree describes the relation between lexical classes and their appropriate superior concepts. This paper describes an implementation of a lexical class identification based on context-free grammars and parsing methods. The semantic analysis of lexical classes is based on grammars enriched with semantic tags. The main algorithm is described together with the experimental results.**

I. Introduction

The human-computer communication is one of the most challenging topics of Artificial Intelligence (AI). Since speech is the most natural and the most common way of human communication, there is a need for using it as a communication medium between human and computer.

The process of extracting the meaning from an utterance by a computer and storing the meaning in a computer model is called Natural Language Understanding (NLU). *Semantic analysis* is the first step of the NLU process. The goal of semantic analysis is to represent what the subject intended to say.

Recent trends in the area of NLU are heading towards making all the processes stochastic. In a stochastic method of semantic parsing, semantic knowledge is usually represented by some kind of a tree.

II. Lexical classes

Lexical class is a word or word group with a similar specific meaning. For instance, in a flight information system, a typical class might be CITY = {'London', 'San Francisco'}. There might also be generic lexical classes which are independent of the domain, e. g. time and date. Given these classes, all words in the sentence are replaced with corresponding lexical class when possible.

The *abstract semantic annotation* describes the relationship between elements and lexical classes in the sentence. It is not necessary to annotate each word from the sentence; words with semantic meaning should be considered particularly. An example of semantic annotation is shown in figure 1. The annotations are mostly created by human annotators. Thus, the process of creating annotated data is very expensive.

M. Konopík is with University of West Bohemia, Department of Computer Sciences, Univerzitní 22, CZ - 306 14 Plzeň, Czech Republic konopik@kiv.zcu.cz

I. Habernal is with University of West Bohemia, Department of Computer Sciences, Univerzitní 22, CZ - 306 14 Plzeň, Czech Republic habernal@kiv.zcu.cz

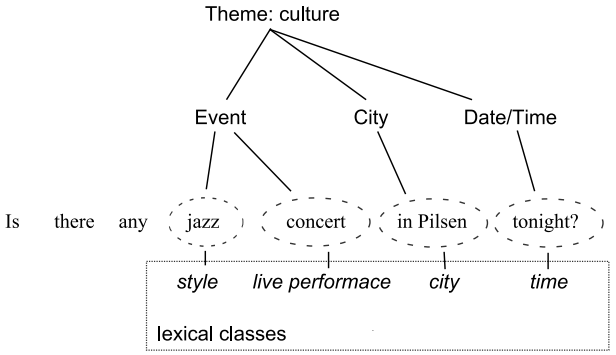

Fig. 1. An example of abstract semantic annotation tree

Identification of lexical classes is then the fundamental step of semantic analysis. Having obtained a set of lexical classes, a tree is then built upon it as shown in figure 1. This tree describes the relation between lexical classes and their appropriate superior concept (e. g. lexical class 'Time' belongs to the concept 'Date/time' and this concept belongs to the 'Theme: culture' superior concept).

III. Lexical class identification

The lexical classes usually have a consistent structure which can be described by context-free grammars [2]. Having obtained a set of annotated sentences with assigned lexical classes (e. g. date and time) created by human annotators, we can develop an appropriate grammar for describing certain lexical classes.

For the identification process, we developed a *local parsing method*. There are various parsing algorithms with different parsing approaches. Since lexical classes are structures consisting mostly of a few words, a bottom-up parser [6] seems to be an efficient tool for this identification.

In this paper, the bottom-up active chart parser is used. It parses a context-free grammar without restriction to any normal form. Moreover, it is still an efficient parsing method [1].

A. Preprocessing

In the initialization part, all the input grammars are loaded. The grammars are hand-written and they are stored in text files in JSGF format [8]. Single grammar describes a single lexical class, hence the parser must gain information about the name of the lexical class and the starting symbol of the grammar.

The input sentence is split into words (tokens). The parser looks up in the grammar whether the token is present on the

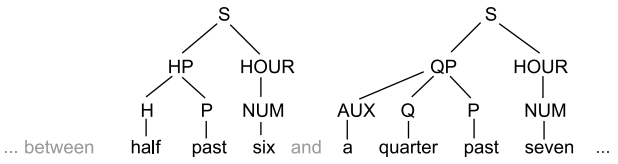

Fig. 2. Successfully parsed lexical classes example.

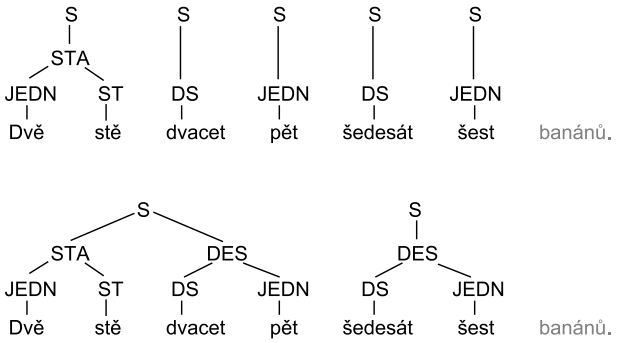

Fig. 3. There are more combinations of possible successful parse trees, as shown above. The parser chooses the two widest trees.

right side of any rule. Tokens which are not a part of the grammar are skipped. This preprocessing phase splits the input sentence into some clusters which are potential lexical classes.

B. Parsing

Having obtained the input string, the parsing process can begin. The parser attempts to build the parse tree over the input words. The parser succeeds if there is a syntactic tree with starting symbol as the root node. Figure 2 shows such a result for grammar describing date and time; the grammar used would be very simple.

Let us have the following input sentence in Czech: *Dvě stě dvacet pět šedesát šest banánu.* (Two hundred and twenty-five sixty-six bananas) and the grammar generating numbers. The phrase makes no sense, but it contains two grammatically correct consecutive numbers – 225 and 60. From another point of view, there can be numbers 200, 20, 5, 60 and 6 (in Czech only). These possibilities are shown in figure 3.

Thus, the parser chooses *all the widest trees from the left that have the starting symbol as the root*. In our case, the tree covering the first four words (the number 225) and the tree covering two remaining words (the number 66) are chosen. The parsing was successful, according to our expectation.

Ambiguity is a major issue in parsing. Let us suppose we have an ambiguous grammar for a certain lexical class. If there are two or more trees from position i to j with the starting symbol as root, the parser cannot decide which tree is the right one without any additional information (probabilities, etc.). However, this is not an issue in the process of identification. We simply say that there is a lexical class between indexes i and j. The ambiguity is undesirable further in semantic analysis of lexical classes.

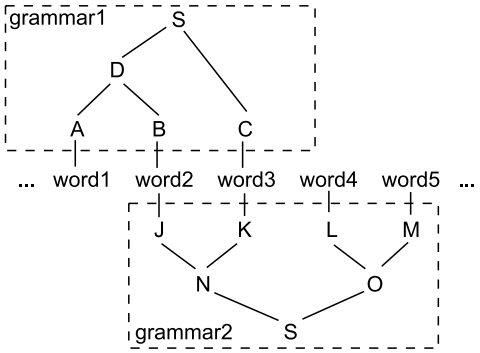

Fig. 4. Two overlapping parse trees for two grammars describing different lexical classes.

Another problem can be an overlap of parsing trees of two different grammars. It means that there are two successful parse trees T_1 and T_2, where T_1 is generated by grammar G_1 and T_2 is generated by grammar G_2. The tree T_1 if from index a to b, the tree T_2 from x to y and $x \leq b$. Again, in this case we cannot decide which lexical class is really present in the sentence and which lexical class has been identified as wrong. This problem is illustrated in figure 4.

IV. Semantic analysis of lexical classes

The method of lexical class semantic analysis is based on handwritten context-free grammars enriched with semantic tags. Associating the rules of a context-free grammar with semantic tags is beneficial; however, after parsing the tags are spread across the parse tree and it is usually hard to extract the complete semantic information from it. Thus, we developed an easy-to-use and yet very powerful mechanism for tag propagation. The mechanism allows the semantic information to be easily extracted from the parse tree. The propagation mechanism is based on an idea to add propagation instruction to the semantic tags. The tags with such instructions are called *active tags* in this paper.

An active tag is a semantic tag enriched with a special processing instruction that controls the process of merging the pieces of semantic information in the parse tree. When the active tags are used and evaluated, the semantics is extracted from the tree in a form of one resulting tag that contains the complete semantic information.

The semantic information from the tree is joined in the following way. Each superior tag is responsible for joining the information from the tags that are placed directly below the superior tag. By a recursive evaluation of the active semantic tags, the information is propagated in a bottom-up manner in the tree.

An active tag contains at least one reference to a sub-tag. During evaluation the reference to a sub-tag is replaced with a value of the sub-tag. The reference has the following notion: #number (e.g. #2). The sub-tags are automatically numbered in the same order as stated in the grammar. Then the number in the sub-tag reference says which tag with a given number will be used to replace the reference.

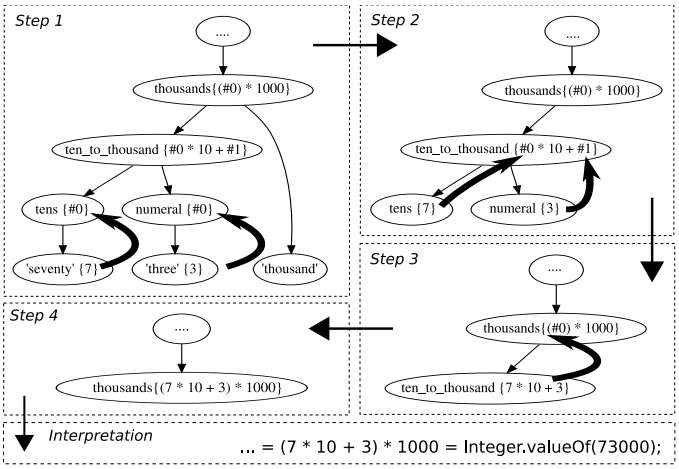

Fig. 5. An example of tag processing for spoken numbers domain

A. Spoken Numbers Analysis Example

This section shows an comprehensive example that demonstrates the processing of spoken number phrases. We reduced the grammar to numbers from 0 to 999 999 because of space limitation.

The semantic grammar with active tags is defined in JSGF as follows:

```
#JSGF V1.0 UTF-8;

<S> = <thousands> {#0} | <hundreds> {#0} |
    <one_to_hundred> {#0};

<thousands> = (<thousand> <hundreds>) {#0+#1} |
    <thousand> {#0} |
    (<thousand> and <one_to_hundred>) {#0+#1};
<thousand> =  (<numeral> thousand) {#0*1000} |
    (<ten_to_thousand> thousand) {(#0)*1000};
<ten_to_thousand> = (<tens> <numeral>) {#0*10+#1} |
    <tens> {#0*10} | <teen> {#0} | <hundred> {#0} |
    (<hundred> <one_to_hundred>) {#0+#1};

<hundreds> = <hundred> {#0} |
    (<hundred> and <one_to_hundred>) {#0+#1};
<hundred> = <numeral> {#0 * 100} hundred;
<one_to_hundred> = (<tens> <numeral>) {#0*10+#1} |
    <tens> {#0*10} | <teen> {#0} | <numeral> {#0};

<tens> = twenty {2} | thirty {3} | forty {4} |
    fifty {5} | sixty {6} | seventy {7} | eighty {8} |
    ninety {9};

<teen> = ten {10} | eleven {11} | twelve {12} |
    thirteen {13} | fourteen {14} | fifteen {15} |
    sixteen {16} | seventeen {17} | eighteen {18} |
    nineteen {19};

<numeral> = one {1} | two {2} | three {3} | four {4} |
    five {5} | six {6} | seven {7} | eight {8} | nine {9};
```

The processing of the phrase "seventy three thousand" is depicted in figure 5. During the tags extraction phase (steps 1 to 4) the resulting tag "$(7*10+3)*1000$" is created. Then it is simply interpreted (in our case with the BeanShell [9]) and the number 73000 is generated.

V. Implementation

The implementation of the program is based on the bottom-up active chart parser implementation described below. It is a command line application written in Java.

A. Bottom-up Chart Parser Implementation

The bottom-up active chart parser algorithm can be described as follows: this algorithm reads input words and attempts to build syntactic trees from the bottom to the top. It uses the *chart* for storing results and the *agenda* for newly created *edges*. Parsing has been successful if there is an edge containing the grammar's starting symbol that covers the whole input [1].

The chart parser implementation is based on our open-source partial Java Speech API implementation. The reason, why our own implementation of JSAPI is used instead of using some third party libraries, is that a tree structure is further used by the parser. Also, the JSAPI specification does not provide any methods for building multiple parsers upon it.

B. Input and output

All input and output files are in the XML format. The format of the input is the same as the format of input file for JAAE [3]. The file contains a list of user queries in which the lexical classes are to be identified.

The output file is another XML file with the following structure (the input sentence is *'Is there any bus number twenty-two from Bory around ten o'clock tomorrow morning?'*):

```
<?xml version="1.0" encoding="utf-8"?>
<sentences>
  <sentence source="Jede mi zejtra kolem deseti
  dopoledne autobus z Bor?">
      <lexicalClass class="time" from="3" to="6">
          kolem deseti dopoledne
      </lexicalClass>
      <lexicalClass class="date" from="2" to="3">
          zejtra
      </lexicalClass>
  </sentence>
  ...
</sentences>
```

The program applies all registered grammars to each input sentence. In the example above, the input sentence contains three lexical classes, each of different type. The source sentence appears in the output as the `source` attribute of the `sentence` element.

The `sentence` element can contain `lexicalClass` subelements. Each lexical class has its appropriate type (the `class` attribute), the starting and the ending position and the content. If there are no lexical classes identified, the `sentence` element is empty.

VI. Experimental results

The lexical classes we have attempted to identify are

- *date interval* — e. g. 'next week', 'this month', etc.,
- *date* — e. g. 'today', 'next sunday', '14th January 2007', etc.,
- *time* — e. g. 'now', 'in the afternoon', 'at six p. m.', etc.,
- *numbers*.

Lexical class	*date*	*date-interval*	*time*	*number*
Sentences count	540	380	466	55
Typing errors	24	15	45	3
Ungrammatical phrases	52	41	72	1
Irrelevant phrases	12	16	3	0

TABLE I
STRUCTURE OF THE INPUT SENTENCES.

Lexical class	*date*	*date-interval*	*time*	*number*
Sentence count	540	380	466	55
Relevant sentence count	504	349	418	52
Recognized sentences	462	330	390	51
Accuracy	86%	87%	84%	93%
Relev. accuracy	92%	95%	93%	98%

TABLE II
IDENTIFICATION RESULTS.

A. Testing data

The original set of input sentences was collected as a part of the LINGVO project[1]. It is a large set of user text queries in Czech with relation to weather information, snow conditions, train and city-bus information, etc. From this set a collection of approximately 600 sentences has been chosen. These sentences contain at least one lexical class we are trying to identify.

The JSGF grammars have been developed using approximately 400 sentences. There are four grammars, each lexical class (described above) has its appropriate grammar.

B. Accuracy

The accuracy of the identification process is one of the most important criteria of the developed method. The *typing errors* in the input sentences play an indispensable role (about 5% for each lexical class). This affects the accuracy ratio negatively. Thus, there is a plan to use a dictionary as the pre-processing phase in the future to avoid these errors.

There are also sentences which contain an *ungrammatical expression* of a lexical class. Sometimes, it can be solved simply by upgrading the grammar. But there are still some cases which are hard to describe by grammars. The *irrelevant phrases* are phrases which describe the certain lexical class but they appear very rarely and they use very uncommon expressions. The structure of the input sentences is shown in table I.

The results of lexical class identification are shown in table I. Each column describes one lexical class.

- **Sentence count** is the count of the input sentences which contain at least one occurrence of certain lexical class.
- **Relevant sentence count** it the count of the input sentences after excluding typing errors and irrelevant phrases.
- **Recognized sentences** means that *the parser found at least one expected lexical class in the sentence*. We could further distinguish sentences with more than on lexical class of the same type, but most of the input sentences contain only one occurrence of a certain lexical class.
- **Accuracy** shows the ratio of all sentences and recognized sentences for each lexical class:

$$\text{Accuracy} = \frac{\text{Count}(\textit{recognized sentences})}{\text{Count}(\textit{all input sentences})} \cdot 100\%$$

[1]See `https://liks.fav.zcu.cz/mediawiki/index.php/Research`

- **Relevant accuracy** row displays the accuracy when sentences containing typing errors and irrelevant phrases are excluded from the input set:

$$\text{Relevant accuracy} = \frac{\text{Count}(\textit{recognized sentences})}{\text{Count}(\textit{relevant input sentences})} \cdot 100\%$$

VII. CONCLUSION AND FUTURE WORK

In this paper, the "proof of concept" for lexical class identification and semantic analysis was presented. The algorithms were successfully implemented and tested. Local parsing methods based on active bottom-up chart parser and context-free grammars seem to be an efficient approach to the lexical class identification. The context-free grammars are able to cover more complicated lexical classes such date and time. For simpler lexical classes, the use of regular expression or dictionary look-up would be a suitable approach.

The application for lexical class analysis has a straightforward implementation. There are many issues that should be solved before the real deployment. Also, the grammars should be written using a larger set of user queries. However, it is apparent that the approach used still yields good results and could be used in real applications after some optimization.

VIII. ACKNOWLEDGEMENTS

This work was supported by grant no. 2C06009 CotSewing.

REFERENCES

[1] James Allen. *Natural Language Understanding (2nd ed.)*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1995.

[2] N. Chomsky. Three models for the description of language. *IRA Transactions on Information Theory*, 2(3):113–124, 1956.

[3] I. Habernal and M. Konopík. *JAAE: The Java Abstract Annotation Editor.* In INTERSPEECH-2007, 1298-1301.

[4] I. Habernal. Lexical Class Analysis. diploma thesis, Department of Computer Science and Engineering, Faculty of Applied Sciences, University of West Bohemia, May 2007.

[5] Y. He and S. Young. Semantic Processing Using the Hidden Vector State Model. *Computer Speech and Language*, 19(1):85–106, 2005.

[6] Daniel Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[7] M. Konopík. *Stochastic Semantic Parsing.* PhD study report, Department of Computer Science and Engineering, Faculty of Applied Sciences, University of West Bohemia, May 2006. Technical Report No. DCSE/TR-2006-01.

[8] Sun Microsystems Inc. *Java Speech Grammar Format Specification* [online]. 1998.

[9] P. Niemeyer. BeanShell 2.0 Documentation, URL: <http://www.beanshell.org/>