Converting First Order Logic into Natural Language: A First Level Approach

Aikaterini Mpagouli, Ioannis Hatzilygeroudis
University of Patras, School of Engineering
Department of Computer Engineering & Informatics, 26500 Patras, Hellas
E-mail: {ihatz, mpagouli}@ceid.upatras.gr

Abstract

In this paper, we present the FOLtoNL system, which converts first order logic (FOL) sentences into natural language (NL) ones. The motivation comes from an intelligent tutoring system teaching logic as a knowledge representation language, where we use it as a means for feedback to the users. FOL to NL conversion is achieved by using a rule-based approach, where we exploit the pattern matching capabilities of rules. The system consists of a rule-based system and a lexicon. The expert system implements the conversion algorithm, which is based on a linguistic analysis of a FOL sentence, and the lexicon provides grammatical information that helps in producing the NL sentence. The whole system is implemented in Jess, a java-based expert system shell. The FOLtoNL is not complete in all its aspects yet.

Keywords : First Order Logic, Natural Language Processing, Expert Systems

1. Introduction

To help teaching the course of "Artificial Intelligence" in our Department, we created a web-based intelligent tutoring system, called Artificial Intelligence Teaching System (AITS) [Hatzilygeroudis etal (2005)]. One of the issues that AITS deals with is the conversion of natural language (NL) sentences into first-order logic (FOL) formulas. Given that this is a non-automated process, it is difficult to give some hints to the students-users during their effort to convert an "unknown" (to the system) NL sentence into a FOL formula. However, some kind of help could be provided, if the system could translate (after checking its syntax) the proposed by the student FOL formula into a NL sentence. The student then will be able to compare the initial NL sentence with the one that its FOL formula corresponds to. In this way, it is easier to see whether his/her proposed FOL formula is compatible with the given NL sentence and perhaps make some amendments, before submitting the final answer.

Although there are research efforts at converting NL sentences to FOL formulas, it seems that there are no such efforts for the inverse process, the conversion of FOL

formulas to NL sentences. The obvious reason is that it is not usually a need, rather the inverse is. However, in our case, this is not the case.

In this paper, we introduce a method for converting FOL formulas into NL sentences, called FOLtoNL conversion. To achieve that, we use the expert systems approach alongside natural language processing aspects.

The structure of the paper is as follows. Section 2 deals with related work. Section 3 presents the basic principles of FOL to NL conversion process. Section 4 refers to the implementation of the FOLtoNL system and presents its architecture. In Section 5, the function of the lexicon of the system is described, whereas Section 6 presents some examples of the system application. Section 7 deals with current restrictions of the systems and give hints for their solution and finally Section 8 concludes the paper.

2. Related Work

In the existing literature, we couldn't trace any directly similar effort, i.e. an effort to translate FOL sentences into natural language sentences. However, we traced a number of indirectly related efforts, those of translating some kind of natural language expressions into some kind of FOL ones.

In [Rodríguez de Aldana (1999)] an application of Natural Language Processing (NLP) is presented. It is an educational tool for translating Spanish text of certain types of sentences into FOL implemented in Prolog. This effort gave us a first inspiration about the form of the lexicon we use in our FOLtoNL system.

In [Fuchs et al (1999)], ACE (Attempto Controlled English), a structured subset of the English language, is presented. ACE has been designed to substitute for formal symbolisms, like FOL, in the input of some systems in order to make the input easier to understand and to be written by the users. This is useful for theorem provers and model builders which are difficult to use for those not familiar with them. ACE expressions are automatically and unambiguously translated in the formal symbolism used as input language for such systems.

In [Bos (2003)], the importance of model building in natural language understanding systems is stressed by explaining how such a process can be used. Model building is used as an intermediate step for understanding natural language sentences.

Finally, in [Pease and Fellbaum (2004)], a Controlled English to Logic Translation system, called CELT, allows users to give sentences of a restricted English grammar as input. The system analyses those sentences and turns them into FOL. What is interesting about it is the use of a PhraseBank, a selection of phrases, in order to deal with the ambiguities of some frequently used words in English like have, do, make, take, give etc. CELT solves ambiguities via WordNet.

3. FOLtoNL Conversion Process

Our FOLtoNL conversion algorithm takes as input FOL formulas of the following form (in a BNF notation, where '[ ]' denotes optional and '< >' non-terminal symbols):

[<quant-expr>] <stmt1> [=>] [<stmt2>].

The formula is in its Prenex Normal Form (i.e. all quantifiers are at the left-hand side of the formula), where <quant-expr> denotes the expression of quantifiers in the formula, '=>' denotes implication and <stmt1> and <stmt2> denote the antecedent and the consequent statements of the implication. <stmt1> and <stmt2> are FOL statements that do not contain any quantifier or implication. So, our algorithm deals with FOL formulas with only one level of implication. Therefore, we call it the *basic FOLtoNL conversion algorithm*. However, it can be easily extended to FOL formulas with more implication levels. For typing convenience, we use the following symbols in our FOL formulas: '~' (negation), '&' (conjunction), 'V' (disjunction), '=>' (implication), 'forall' (universal quantifier), 'exists' (existential quantifier).

For example, the following FOL formula is a proper input to the algorithm:

```
(forall x)(exists y)human(x)&mother(y,x)=>loves(y,x)
```

where <quant-info> ≡ "`(forall x)(exists y)`", <stmt1> ≡ "`human(x)& mother(y,x)`", <stmt2> ≡ "`loves(y,x)`".

The output of the algorithm has the following structure:

[<interpret-quant-info>] [**if**] <interpret-stmt1> [**then**] [<interpret-stmt2>]

where <interpret-quant-info> denotes the interpretation of quantifiers, <interpret-stmt1> and <interpret-stmt2> the interpretation of <stmt1> and <stmt2> respectively in NL and 'if', 'then' are terminal symbols.

The high-level description of our algorithm is as follows:

1. Scan the user input and determine <quant-info>, <stmt1> and <stmt2>
2. if <quant-info> ≠ ∅
   2.1 if <stmt2> ≠ ∅, the output has the form
   <interpret-quant-info> **if** <interpret-stmt1> **then** <interpret-stmt2>
   2.2 the output has the form
   <interpret-quant-info> <interpret-stmt1>
3. the output has the form
   <interpret-stmt1>

In the sequel, we explain how <interpret-quant-info>, <interpret-stmt1> and <interpret-stmt2> are determined.

3.1 Interpretation of Quantifiers

We distinguish four cases of quantifiers: (a) universal quantifier, (b) negated universal quantifier, (c) existential quantifier and (d) negated existential quantifier. Also, we denote by u_i, nu_i, e_i and ne_i the variables bound to the previous types of quantifiers respectively.

The algorithm for quantifiers interpretation is as follows:

1. Set <interpret-univer-quant-info> ≡ ∅, <interpret-exist-quant-info> ≡ ∅, <interpret-neg-exist-quant-info> ≡ ∅
2. If <quant-info> contains both universal quantifiers and negated universal quantifiers, <interpret-univer-quant-info> ≡ "**for all** $u_1, u_2, \ldots, u_n$ **and for some but not all** $nu_1, nu_2, \ldots, nu_m$"
3. If <quant-info> contains only universal quantifiers, <interpret-univer-quant-info> ≡ "**for all** $u_1, u_2, \ldots, u_n$"
4. If <quant-info> contains only negated universal quantifiers, <interpret-univer-quant-info> ≡ "**for some but not all** $nu_1, nu_2, \ldots, nu_m$"
5. If <quant-info> contains existential quantifiers,
   5.1 if the number of bound variables is one, <interpret-exist-quant-info> ≡ "**exists** e_1 **such that**"
   5.2 else, <interpret-exist-quant-info> ≡ "**exist** $e_1, e_2, \ldots, e_l$ **such that**"
6. If <quant-info> contains negated existential quantifiers,
   6.1 if the number of bound variables is one, <interpret-neg-exist-quant-info> ≡ "**there is no** ne_1 **such that**"
   6.2 else, <interpret-neg-exist-quant-info> ≡ "**there are no** $ne_1, ne_2, \ldots, ne_r$ **such that**"
7. <interpret-quant-info> ≡ <interpret-univer-quant-info> <interpret-exist-quant-info> <interpret-neg-exist-quant-info>

3.2 Interpretation of antecedent and consequent statements

We assume that the antecedent and consequent statements of an implication (<stmt1> and <stmt2>) have the following format:

<atom$_1$> <con$_{12}$><atom$_2$><con$_{23}$><atom$_3$>…<com$_{(n-1)n}$><atom$_n$>

where <con$_{ij}$> is one of the connectives 'V' and '&'. However, at the present version an antecedent or consequent statement cannot have both 'V' or '&'. The interpretation of such a statement has the following format:

<interpret- atom$_1$> <interpret- con$_{12}$><interpret- atom$_2$>…< interpret- atom$_n$>

where <interpret- con$_{ij}$> ≡ "or", if con$_{ij}$ ≡ "V", and <interpret- con$_{ij}$> ≡ "and", if con$_{ij}$ ≡ "&".

In the present version, we consider only atoms with predicates of arity one and two, called *single-term* and *two-term* atoms respectively. We further distinguish between *plain* and *negated* atoms. Interpretation of a single-term atom is presented in Table 1. Furthermore, interpretation of two-term atoms is presented in Table 2. The symbol **P** means that predicate P is modified as far as its ending is concerned to be compatible with the verb expression it is associated with. Interpretation of constants and variables is the constants and variables themselves. Interpretation of function expressions is presented in Table 3. Notice that we consider function expressions with one or two terms only that are no other function expressions.

Table 1. *Interpretation of single-term atoms*

P	P(t)	~P(t)
verb	<interpret-t> **does P**	<interpret-t> **does not P**
noun	<interpret-t> **is a P** <interpret-t> **is an P**	<interpret-t> **is not a P** <interpret-t> **is not an P**
adjective	<interpret-t> **is P**	<interpret-t> **is not P**

Table 2. *Interpretation of two-term atoms*

P	$P(t_1, t_2)$	$\sim P(t_1, t_2)$
verb	<interpret-t_1> **P** <interpret-t_2>	<interpret- $_1$> **does not P** <interpret-t_2>
noun	<interpret-t_1> **is the P of** <interpret-$_2$>	<interpret-t_1> **is not the P of** <interpret-t_2>

Table 3. *Interpretation of function expressions*

t	<interpret-t_i>	
	f(t)	$f(t_1, t_2)$
noun	**the f of t**	**the f of t_1 and t_2**
adjective	**the f t**	

To be able to recognize which type of speech is a predicate or a term, in order to apply the rules of Tables 1-3, we created a lexicon. The lexicon is actually a base of facts and is presented in the next sections.

4. Implementation Aspects

The FOLtoNL process has been implemented in Jess [Friedman-Hill (2003)]. Jess is a rule-based expert system shell written in Java, which however offers adequate general programming capabilities, such as definition and use of functions. The architecture of the implemented system is presented in Figure 1.

The system includes two Jess modules, MAIN and LEX. Each Jess module has its own rule base and its own facts and can work independently from the rest of Jess modules. Focus is passed from one module to the other to execute its rules. MAIN is the basic module of the system, whereas LEX is the system's lexicon.

Apart from information about English words, the LEX module also includes two rules for their treatment. The rule **part-of-speech** is activated when a fact of the form (lemma ?lem) is inserted in LEX. This kind of fact declares that MAIN module needs information for the English word '?lem' from the lexicon. When, subsequently, the focus is given to LEX the rule fires, looking for the specified word. Provided that the word is found, the rule updates the global variable '?*part*' with the word's 'part of speech'. The rule **expression** is used for words with a special syntax. This rule returns the syntax of such words to MAIN and makes it possible to appear in the correct from in the corresponding interpretation.

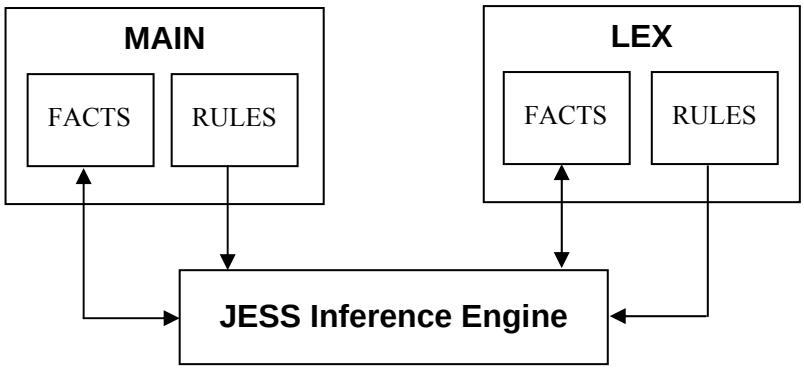

Figure 1. *System Arcitecture*

Jess uses the notion of templates, which are structured descriptors, consisted of slot-value pairs, for describing complex facts. We have defined five templates in the MAIN module: Atom, Term, Function, SimpleStatement and Statement. Apart from its predicate and its terms, each atom (Atom) is characterized as plain or negated. It is also characterized by the statement it belongs to (the antecedent or the consequent). Each statement (SimpleStatement) consists of a list of the atoms that constitute it and the connective between them. The order of the atoms in each statement's list is the same as the order in which they occur in the input formula (sentence). The sentence (Statement) represents the whole formula given as input and consists of two statements (SimpleStatements), the antecedent and the consequent. Apart from each template's individual slots, all templates have one common slot, 'interpretation', which is filled in with the corresponding interpretation, when it is available.

MAIN's most important function is *readinput*, which takes the user's input as an argument and processes it as a string. The result is to update the global vectors

representing the quantifiers, the statements (antecedent and consequent) and the connectives with corresponding values (i.e. bound variables, constituent atoms and connectives). Then, *readinput* calls the function *read-atom* for atom analysis. For each atom an appropriate Atom fact is created. For identification and interpretation of the terms of each atom, *read-atom* calls *read-term*, *atom-interpret* and *function-interpret* (when the term is a function). To this end, function *word-pos* is employed, which takes as an argument a word (representing either a function symbol or a predicate) and returns an one-character symbol, that shows which is the type of the word (i.e. noun, verb or adjective) and a string, which is the appropriate form of the word (e.g. with changes in its ending), if necessary. This refers to **P** in Tables 1 and 2.

Two other interpretation functions are *simple-interpret* and *statement-interpret***,** for the two statements and the final output respectively. The function *simple-interpret* interprets a statement based on the interpretations of its atoms and the connective associated with them. The function *statement-interpret* uses the value of the global vectors representing the quantifiers and their bound variables and the interpretations of the statements to update the slot 'interpretation' of Statement. The value of this slot is the output of the system.

After *readinput* has finished its operation, one or two SimpleStatement facts and a Statement fact have been created. The Atom, SimpleStatement and Statement facts have their 'interpretation' slots empty, waiting for their values. The values are deduced via MAIN rules, which call the interpretation functions we mentioned above. First, all the atoms are interpreted via the rule **atom-interpret**. After all atoms have had their 'interpretation' slots updated, statements are interpreted via the rule **simple-interpret**. After statements have been interpreted, interpretation of the whole formula takes place via the rule **statement-interpret**.

5. The Lexicon

The lexicon consists of a large number of facts concerning words, called *word-facts*. The lexicon was built using an existing text file (LEX.txt), which contains all English words: http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/areas/nlp/corpora/keiras. The file includes one line string for each English word. From each line's string we use 26 characters. Each of the first 6 characters denotes a different characteristic of the word: 'part of speech' (pos), 'past participle' flag, 'negative' flag, 'to be' flag, 'verb + ing' flag and 'aux' flag. Characters 7-26 are left for the English word itself.

So, we transformed each line to an appropriate fact, keeping the extracted information (from LEX.txt) and adding some new characteristics (?exp, ?gen etc). We also removed words that are very unlikely to appear in FOL sentences. This way we reduced the final lexicon's size. A small part of the Lexicon facts are presented in Figure 2. Each word-fact is an instance of the following template:

(word ?type ?gen ?form ?exp ?lemma ?lem)

where 'word' declares that it is a fact describing a word and the rest are variables representing the fields that describe the word, which are explained below:

- **?type:** denotes the type of the word and takes one of the following values:
- **?gen:** denotes the gender of the word and takes one of the following values:
- **?form:** denotes the form (singular or plural for nouns and adjectives, person for verbs) and takes one of the following values:
- **?exp:** denotes the existence of a special syntax of the word; if such syntax exists; it's an index to a fact of the 'expression' template and '0' otherwise.
- **?lemma:** is the form by which we look up the word in a lexicon.
- **?lem:** the word that the fact is about.

```
(deffacts lexicon

        (word J 0 0 a1 able able)
        (word J 0 0 0 abnormal abnormal)
        (word V 0 1 0 accept accepts)
        (word J 0 0 0 artificial artificial)
        (word V 0 0 a1 appeal appeal)
        (word V 0 1 a1 appeal appeals)
              .....
)
```

Figure 2. *Lexicon sample*

6. System Application

In Figure 3, an example use of the FOLtoNL system is illustrated, which reveals the current user interface of the system.

```
Jess> (batch FOLtoNL.clp)

First Order Logic to Natural Language
Give a sentence in First Order Predicate Calculus:

(forall x)(exists y)loves(x,y)

INTERPRETATION : forall x exists y such that x loves y .
```

Figure 3. *An example use of the FOLtoNL system*

In Table 4, a number of input-output pairs to the FOLtoNL system are presented.

7. Future Work

The FOLtoNL system has a number of restrictions that need further development. First, FOLtoNL gives interpretations into a NL-like language that is closer to FOL than to pure NL. This requires a second level of interpretation, above the present one.

Present level of FOLtoNL conversion has also its own problems. It has a problem when interpreting sentences which are entirely in the scope of a negation. This means that while the system can deal with '~', when it appears in front of a single atom or a quantifier, it can't deal with it, when it has an entire sentence or statement in its scope. As a consequence, the user bears the burden of appropriately reforming his input. A possible solution would require some changes in the *readinput* function, to be able to store extra information.

Table 4. *Example interpretations of FOL sentences using FOLtoNL*

Input	Output
(forall x)(forall y) likes(x,y) => appeals(y,x)	**forall x and y if x likes y then y appeals to x**
(forall x) bat(x) => ~feathered(x)	**forall x if x is a bat then x is not feathered**
(forall x)(loves(father(x),x)&loves(mother(x),x))	**forall x the father of x loves x and the mother of x loves x**
~(forall x)(exists y)cares(x,y)	**for some but not all x , exists y such that y cares about x**
is(sum(2,3),5)	**the sum of 2 and 3 is 5**
(forall x)(exists y)(exists z)(exists w) human(x) => name(y,x) & age(z,x) & birthday(w,x)	**forall x exist y , z and w such that if x is a human then y is the name of x and z is the age of x and w is the birthday of x**
(forall x) bird(x) & ~flies(x) &swims(x) => penguin(x)	**forall x if x is a bird and x does not fly and x does swim then x is a penguin**

Another constraint is forced upon the use of '=>', which can only occur once in the input sentence. So, we cannot use more than one-level implications in the input sentence, such as "(stmt1 => (stmt2 => stmt3))". This problem could be solved using an extra template concerning implication.

Another restriction is that the connectives in each sub-sentence must be of the same kind (either '&' or 'V'). The truth is that we can use different connectives, but in that case the system will give an ambiguous output. For example, for "if <a> and <b> or <c>..." we can't tell if the initial sentence was "(a & b) V c…" or "a & (b V c)…". This information could be given in the output via the appropriate use of comas.

Furthermore, the atoms and the functions can have only one or two terms. Hence, relations between three or more entities can't be expressed. Moreover, the system can't interpret functions with other functions as their terms. To solve this problem, changes to the *read-atom* function should be made.

Another problem has to do with adjectives in comparative or superlative form. The system cannot discriminate the different forms of an adjective, so uses the same

syntax for those different forms. This problem can be solved if we include such extra information in the lexicon.

Finally, another limitation concerns the lexicon of the system. It currently contains a limited number of words. It should be extended to include as more as possible.

8. Conclusions

In this paper, we present the FOLtoNL system, which converts first order logic (FOL) sentences into natural language (NL) ones, to help students. FOL to NL conversion is achieved by using a rule-based approach, where we exploit the pattern matching capabilities of rules. The system consists of a rule-based expert system and a lexicon. The expert system implements the conversion algorithm, which is based on a linguistic analysis of a FOL sentence, and the lexicon provides grammatical information that helps in producing the NL sentence. The whole system is implemented in Jess, a java-based expert system shell. We are not aware of other similar efforts, although there are efforts to deal with the inverse problem. Neither of them, however, uses an expert systems approach.

Regarding the usefulness of our system, the first reactions of the students were encouragingly positive. We need, however, a more systematic user study confirming our intuition about the system's usefulness and this is one of our future goals. Also, as illustrated in Section 7, our effort has not been completed yet. The FOLtoNL system has a number of restrictions that are challenges for further work.

References

Bos J. (2003), *Exploring Model Building for Natural Language Understanding*, in Proceedings of the 4th Workshop on Inference in Computational Semantics (ICoS-4), 25-26 September, 2003, Nancy, France (available at http://www.iccs.informatics.ed.ac.uk/ ~jbos/pubs/bos-icos4.pdf).

Friedman-Hill E. (2003), *Jess in Action: Rule-Based Systems in Java*, Manning Publishing, 2003.

Fuchs N. E., Schwertel U., Torge S. (1999), *Controlled Natural Language Can Replace First Order Logic*, in Proceedings of the 14th IEEE International Conference on Automated Software Engineering (ASE'99), pp. 295-298 (available at http://www.ifi.unizh.ch/ groups/req/ftp/papers/ASE99.pdf).

Hatzilygeroudis I., Giannoulis C., Koutsojannis C. (2005), *Combining Expert Systems and Adaptive Hypermedia Technologies in a Web Based Educational System*, Proceedings of the IEEE ICALT-2005, Kaohsiung,Taiwan, July 5-8, 2005, pp 249-253.

Pease A., Fellbaum C. (2004), *Language to Logic Translation with PhraseBank*, in Proceedings of the Second Global Conference (GWC'04), January 20-23, Brno, Czech Republic, pp. 187 –192.

E. Rodríguez Vázquez de Aldana (1999), *An application for translation of Spanish sentences into First Order Logic implemented in Prolog*, (available at http://aracne.usal.es/ congress/PDF/EmilioRodriguez.pdf).