STUDIA UNIV. "BABEŞ–BOLYAI", INFORMATICA, Volume **XLIV**, Number 2, 1999

SEMANTIC REPRESENTATION OF THE QUANTITATIVE NATURAL LANGUAGE SENTENCES

ADRIAN ONET AND DOINA TATAR

Abstract. In this paper we propose the description of two methods of quantitative natural language sentences representation with the use of the first order logic. Both of these methods extend the classic first order logic. One of these two methods was introduced by J. Allen [4] who extends the classic logic by admitting sets as objects (method 2). The other method, more restrictive, (introduced by the authors of this paper) extends the quantifier semnification - specially the existential one - thus the quantifier would be able to express exact quantities (method 1). This method is a specialization of Allen's method.

In the second part of this paper we describe an implementation of the first method by the lambda calculus and also the transformation of these expressions in first order formulas using Prolog predicates.

1. First order predicate logic as representation language for quantitative natural language sentences

Among the most used methods in natural language representation is the classic first order predicate logic, but unfortunately not all natural language sentences can be represented using the classic first order predicate logic. This is the case of quantitative sentences. For instance, a sentence like "Three men entered the room" is quite difficult to represent by using the first order logic, because there are more quantifiers in natural language that the universal and existential quantifiers. A representation like:

$$\exists X.(men(X) \wedge (\exists Y.room(Y) \wedge enter(X,Y))$$

would not mean the same thing semantically speaking, there is no information in this formula concerning the number of persons which entered the room.

We will describe two methods of extension for the classic first order logic in order to allow the representation of the quantitative aspects in natural language. The first method is weaker (it does not allow the representation of all quantitative aspects) but its implementation is easier (one of its implementations is given in the second part of this paper), and a second method which extends the representation ontology in order to allow sets as objects.

1.1. **First method.** Let $L = (\Sigma, F, A, R)$ be a first order predicate logic [3]:

$$\Sigma = V \cup C \cup (\cup F_i) \cup (\cup P_i) \cup \{(,), \forall, \exists, \wedge, \vee, \rightarrow\}$$

where V represents a set of symbols called variables, C represents a set of symbols called constants, F_j represents a set of function symbols with the arity j, P_j represents a set of predicate symbols with the arity j, $\{\neg, (,), \wedge, \vee, \rightarrow\}$ represents logic operators and $\exists$ and $\forall$ represent the logic quantificators, F represents a set of well-formed formulas, A represents a set of axioms over L and R a set of rules over L.

We can extend this logic by introducing a pair (M, R) where M represents a set of numbers and R represents the order relation "$<$" over M. If E is the domain of the logic and $|E|$ is finite then M can be the set $\{1, ..., n\}$ where $|E| = n$. If E is infinite then we will create M such as $|M| = |E|$.

We introduce a new set of quantificators, $\{\exists_n | n \in M\}$ where $\exists_n x.(p(x))$ means that there exists exactly n x in E such that $p(x)$. If we introduce the exclusive or operator $\otimes(p \otimes q \leftrightarrow (\neg p \wedge q) \vee (p \wedge \neg q))$ we have:

$$\exists x.p(x) \leftrightarrow \otimes_{j \in M} \exists_j x.p(x),$$

and it can also be proved that $\exists_n x.p(x) \rightarrow \exists x.p(x)$ is a theorem $\forall n \in M$.

In addition, we introduce the following predicates to describe the relation between the elements of M:

$p < q \leftarrow R_{pq}$
$greater(p, q) \leftarrow \neg p < q$
$most(p) \leftarrow |E|/2 < n$
$some(n)$ iff $n \in M$

In actual discourse, the interpretation of the most will usually be relative to some previously defined context. The most predicate can be rewritten to represent the most concept in the natural language.

With this extension, a sentence like "Three men enter the room" would be interpreted like

$$\exists_3 X.(man(X) \wedge (\exists_1 Y.room(Y) \wedge enter(X, Y))),$$

where the first quantificator $\exists_3$ means that there is exactly 3 men who enter the room and the second quantificator $\exists_1$ means that the three men enter exactly one room.

With this extension of the first order logic we can also represent inconsistent knowledge expressed in natural language sentences. A sentence like "At least five men walk" has the following interpretation:

$$\exists_n X.(man(X) \wedge (walk(X)) \wedge greater(n, 3)).$$

The disadvantage of this method is that it cannot represent sentences that involve the interaction of two (or more) elements of the quantified set (for example

"Three men meet at four" – we will see that this kind of sentence can be represented using the second method).

Even with this disadvantage this method can be used successfully to represent quantitative sentences thanks to the easy implementation (see section 2).

1.2. **Second method.** This method has been introduced by J. Allen [4] and extends the ontology of the first order predicate logic to allow sets as objects. While sets in general may be finite (such as the set consisting of John and Mary) or infinite (such as the set o numbers greater then 7). In this method we will only use finite sets. We also allow constants to denote sets. Thus S_1 might be the set $S_1 = \{\text{John, Mary}\}$. The sets will be written in the form $\{y|P(y)\}$, which is the set of all object that satisfy the expression $P(y)$. The set of all men is $\{y|Man(y)\}$. In addition, we introduce the following predicates to relate sets and individuals:

$$\begin{array}{ll} S1 \subset S2 & \text{iff all elements of } S_1 \text{ are in } S2 \\ x \in S & \text{iff } x \text{ is a member of the set } S \end{array}$$

With setlike objects in the representation, we can produce an interpretation for "Some men meet at four" as follows:

$$\exists M : (M \subset \{x|Man(x)\} \wedge Meet(M, 4PM))$$

that is, there is a subset of men M that met at three. In principle, sets are allowed in all situation where individuals have been allowed. In practice certain verbs require only sets or only individuals in certain argument positions. For example, the verb meet requires its agent to be a set with more than one element, as a single individual cannot meet. Other verbs require individuals and exclude sets, and others allow both sets and individuals as arguments.

Consider the different formulas that arise from the collective/distributive readings. There are two representation of the sentence "Some men bought a suit". The collective reading would map to:

$$\exists M1 : (M1 \subset \{z|Man(z)\} \wedge \exists_s : (Suit(s) \wedge Buy1(M1, s)))$$

that is there is a subset of the set of all men who together bought a suit. The distributive reading involves some men individually buying suits and would be represented by:

$$\exists M2 : (M2 \subset \{z|Man(z)\} \wedge \forall m : (m \in M2 \wedge \exists s : (Suit(s) \wedge Buy1(m, s))))$$

Note that with the first method described earlier we have:

$$\exists_n X.(man(X) \wedge (\exists_1 Y.(suit(Y) \wedge buy(X, Y)))),$$

which describes the distributive reading.

Note that the distributive and collective readings both involve a common core meaning involving the subset of men. The only difference is weather you use the set as a unit or quantify aver all members af the set.

The set-based representation can also be used to ensure that more that one man bought a suit. To do this we introduce a new function that returns the cardinality of set. For any given set S, let $|S|$ be the number of elements in S. Using arithmetic operators, we can now encode constraints on the size of sets. For example, the meaning of "Three men entered the room" would be as follows (with tense information omitted):

$$\exists M : ((M \subset \{y|man(y)\} \wedge |M| = 3) \wedge \forall m : (m \in M \wedge enter(m, room)))$$

by changing the restriction to $|M| \geq 3$, we get the meaning of "At least three men entered the room", and so on. More problematic quantifiers can also be given an approximate meaning using sets. For example we can use the most definition from the first method (i.e. most is true if more than half of some set has a given property), then "Most men smoke" might have the meaning:

$$\exists M : ((M \subset \{y|man(y)\} \wedge |M| \geq \frac{|\{y|man(y)\}|}{2}) \wedge \forall m : (m \in M \wedge smoke(m)))$$

In actual discourse, the interpretation of the quantified terms will usually be relative to some previously defined set. For example the sentence "Most men smoke" typically will refer to most of the men in a previously mentioned set rather than to most of the men in the world. In other words, the sentence would not claim that more than half of all men smoke, but that more than half the men in a certain context (say in a railway station) smoke.

If we compare the previous two methods, we can observe that the second method extends the first one, i.e. the first method works just at cardinality level of the sets. That is the first method specifies only those aspects which can be represented with the cardinality function in the second method, and the elements of the set doesn't interact with each other. For example the sentence "Mary eat two apples" can be represented using both methods, with the first method, its interpretation will be:

$$\exists_2 X.(apple(X) \wedge eat(mary, X),$$

using the second method the sentence will be mapped as follows:

$$\exists M : (M \subset \{y|apple(y)\} \wedge |M| = 2) \forall m : m \in M.eat(mary, m)$$

But the following sentence "Four men meet" have the following interpretation using the second method:

$$\exists M : (M \subset \{x|Men(x)\} \wedge |M| = 3) \wedge Meet(M),$$

but it can not be represented using the first method, because this method doesn't introduce the set concept. In the next section we introduce an automate process which associates semantic representation (first order predicate logic with the extension introduced by the first method) for quantitative natural language expression.

2. An automate process for associating semantic representation of quantitative natural language sentences

In this section we will use the lambda calculus as a national extension of first order logic with the extension discussed in the first method and then we will give an implementation of this lambda calculus in Prolog. In this section we use the notation given by P. Blackburn and J. Bos in [2,3].

2.1. **The Lambda Calculus.** The lambda calculus is a natural extension of the first order logic that allows us to bind variable using a new binding operator λ. Occurrences of variables bound by λ should be thought of as placeholders for missing information. An operation called β-conversion performs the required substitutions. The lambda operator marks missing information by binding variables. Here is a simple lambda expression:

$$\lambda x.man(x)$$

Here the prefix $\lambda x.$ binds the occurrence of x in $man(x)$. In this example, the binding of the free x variable in $man(x)$ explicitly indicates that man has an argument slot where we may perform substitutions.

Concatenation indicates that we wish to perform substitution. We're using a special symbol "@" to indicate concatenation. The following expresion:

$$\lambda x.man(x)@vincent,$$

yields $man(vincent)$.

The lambda expressions $\lambda x.man(x)$ and $\lambda y.man(y)$ are equivalent, all these expressions are functors which when applied to an argument, replace the bound variable by the argument. No matter which argument A we choose, the result of applying any of the two expression to A and then β-converting is $man(A)$. The process of relabeling bound variables is called α-conversion. Consider the following grammar:

$$\begin{array}{lcl} s & \rightarrow & np, vp. \\ np & \rightarrow & det, noun. \\ det & \rightarrow & 'every'. \\ noun & \rightarrow & 'men'. \\ vp & \rightarrow & 'cry'. \end{array}$$

Now we can build the semantic representation for our first sentence "Every man cries". For this purpose we assign lambda expressions to different basic syntactic categories, i.e.

$$\begin{array}{l} 'every' : \lambda P.\lambda Q.\forall x.(P@x \rightarrow Q@x) \\ 'man' : \lambda Y.man(Y) \\ 'cries' : \lambda X.cry(X) \end{array}$$

According to our grammar, a determiner and a common noun can combine to form a noun phrase. For our analysis we will associate the NP node with the

Figure 1

functional application that has the determiner representation as functor and the noun representation as argument, and then associate that with the verb phrase (VP). Using a graphic representation this will look as the Figure 1 presents.

Now let's take a look at how we represent quantitative sentences with lambda calculus.We will give three examples of DCG (which will bind four general situations) to do this transformation for particular quantitative sentences (according to the first method described in the previous section).

2.1.1. *Definite quantity sentences.* For the definite quantity sentence we will have to represent with the lambda calculus the $\exists_n$ quantifier. This quantifier is mapped by the following predicate: $exists(n, X, formula(X))$ where n represents the quantity (as it's described in the first method), X is a variable and $formula(X)$ is a formula which contains the variable X. The representation for the $\forall$ quantifier is $forall(X, formula(X))$, where X and $formula(X)$ have the same meaning as it was shown.

Lambda expressions will be represented in Prolog as follows: `lambda(L,F)`, where `L` is intended to be a Prolog variable, while `F` is a first order formula or the Prolog representation of a lambda expression. For example the lambda expression $\lambda A.men(A)$ will look in Prolog: `lambda(A,men(A))`. The concatenation will be represented in Prolog with the Prolog operator @ defined as:

```
:- op(950,yfx,@).
```

The indefinite sentences are given by the following determinants: one, two, three, four, ..., three and half, etc. Each of this determinant will be replaced by an expression $\exists_n$ where n represents the number denoted by the determinant. For example $\exists_{3.5}$ represents the determinant three and half. Let be the following sentence "Three men enter the room". In the following we describe a DCG who will take such sentences (which describe definite quantities) and will give us the

lambda expression corresponding to that sentence:

$$s(NP@VP) \rightarrow np(NP), vp(VP).$$
$$np(Det@Noun) \rightarrow det(Det), noun(Noun).$$
$$vp(NP@IV) \rightarrow iv(IV), np(NP).$$

with lexical entries

$$noun(lambda(A, man(A)) \rightarrow [men].$$
$$iv(lambda(B, lambda(C, enter(C, B))) \rightarrow [enter].$$
$$noun(lambda(C, room(C)) \rightarrow [room].$$
$$det(lambda(D, lambda(E, exists(3, X, D@X\&E@X)))) \rightarrow [three].$$
$$det(lambda(F, lambda(G, exists(1, X, D@X\&E@X)))) \rightarrow [the].$$

where $enter(A, B)$ express that A enter in B.

The construction of the first three lexical entries are obvious. Let's now have a look at the construction of the fourth clause:

$$det(lambda(D, lambda(E, exists(3, X, D@X\&E@X)))) \rightarrow [three].$$

These constructions yield that for any determinant which indicates a number, the lambda expression corresponds to $\exists_n$ quantifier where n represents that number (in our case n=3). In the last clause the determinat "the" denotes quantity equal to 1, so it is processed like its predecessor clause.

So if we have the following goal (in Prolog):

```
?-np(Sem,[three men enter the room],[]).
```

the `Sem` will be bounded to (we used bracket for a clear see):

```
Sem=(lambda(D,lambda(E,exists(3,X, D@X&E@X)))@lambda(A,man(A)) ) @
   (lambda(F,lambda(G,exists(1,X, F@X&G@X)))@lambda(C,room(C))) @
   lambda(C,lambda(B,enter(B,C)))
```

In last section we will give a method to apply the β-conversion to a lambda expression and transform to a first order formula. Next we will show this mechanism of β-conversion on the expression `Sem`.

```
1. Sem=(lambda(D,lambda(E,exists(3,X, D@X&E@X)))@lambda(A,man(A)) ) @
   ((lambda(F,lambda(G,exists(1,X, F@X&G@X)))@lambda(C,room(C))) @
   lambda(C,lambda(B,enter(B,C))))
2. Sem=(lambda(E,exists(3,X, lambda(A,man(A))@X&E@X))) @
   ((lambda(G,exists(1,X, lambda(C,room(C))@X&G@X))) @
   lambda(C,lambda(B,enter(B,C))))
3. Sem=(lambda(E,exists(3,X, man(X)&E@X))) @
   ((lambda(G,exists(1,X, room(X))&G@X))) @
   lambda(C,lambda(B,enter(B,C))))
4. Sem=(lambda(E,exists(3,X, man(X)&E@X))) @
   ((exists(1,X, room(X)& lambda(C,lambda(B,enter(B,C)))@X))))
5. Sem=(lambda(E,exists(3,X, man(X)&E@X))) @
   ((exists(1,X, room(X)& lambda(B,enter(B,X))))))
   // with an ?-conversion we will transform X in Y
6. Sem=(lambda(E,exists(3,X, man(X))&E@X))) @
   ((exists(1,Y, room(Y)& lambda(B,enter(B,Y)))))
```

```
7. Sem=(exists(3,X, man(X)& (exists(1,Y, room(Y)& lambda(B,enter(B,Y))) @X)))
8. Sem=(exists(3,X, man(X)& (exists(1,Y, room(Y)& enter(X,Y)))
```

which represents the formula:

$$\exists_3 X.(man(X) \wedge \exists_1 Y.(room(Y) \wedge enter(X,Y)))$$

which is (in the Skolem normal form)

$$\exists_3 X.\exists_1 Y.(man(X) \wedge room(Y) \wedge enter(X,Y))$$

With this method we can represent sentences like "John eats one and half apple" and "The suit costs 400$". But it cannot represent sentences like "Two men look the same" and "Four kids fight with each other".

2.1.2. *Indefinite quantity sentences.* In this part we'll describe how indefinite quantity sentences can be represented using the lambda calculus. The indefinite sentences are given by the following determinants: most, number of, lot of ... For each of this determinant we must construct a special predicate like the most predicate which we defined in the previous section.

Let be the following sentence "Most people laugh". In the following we'll describe a DCG who will take such sentences (which describe indefinite quantities) and will give us the lambda expression corresponding to that sentence:

$$s(NP@VP) \rightarrow np(NP), vp(VP).$$
$$np(Det@Noun) \rightarrow det(Det), noun(Noun).$$
$$vp(V) \rightarrow v(V).$$

with the lexical entries

$$noun(lambda(A, people(A)) \rightarrow [people].$$
$$v(lambda(B, laugh(B))) \rightarrow [laugh].$$
$$det(lambda(C, lambda(D, exists(N, X, C@X\&D@X\&most(N))))) \rightarrow [most].$$

where the predicate *most* is defined in the previous section and $laugh(B)$ express that B laugh. Note that in this case (i.e. for indefinite quantity) we need a variable N instead of a constant. N will denote the number and the predicate most will tell us if that number is enough to represent the most concept.

So if we have the following goal (in Prolog):

```
?-np(Sem,[Most people laugh],[]).
```

the `Sem` variable will be bounded to:

```
Sem=( lambda(C,lambda(D,exists(N,X,C@X&D@X\&most(N))))@
                lambda(A,people(A))@lambda(B,laugh(B))
```

The β-conversion on the expression `Sem` will be:

```
1. Sem=(lambda(C,lambda(D,exists(N,X,C@X&D@X\&most(N))))@
                  lambda(A,people(A))@lambda(B,laugh(B))
2. Sem=(lambda(D,exists(N,X,people(X)&D@X&most(N))) @lambda(B,laugh(B))
3. Sem=(exists(N,X,people(X)&laugh(X)&most(N))
```

which represents the formula:

$$\exists_N X.(man(X) \wedge laugh(X) \wedge most(N))$$

With this method we can represent sentences like "A number of people reads", "A lot of apples are green", etc. But we cannot represent sentences like "Most people meet at three".

2.1.3. *Restrictive quantity sentences.* The restrictive sentences are given by the following determinants: at least, minimum, maximum, at the most. For each of this determinant we must construct a special predicate like the $least/2$ predicate which we defined as follows:

$$Least(N, M) \leftarrow N > M.$$

Let be the following sentence "At least four men cry". The DCG is:

$$s(NP@VP) \rightarrow np(NP), vp(VP).$$
$$np(Det@Noun) \rightarrow det(Det), noun(Noun).$$
$$det(NP@Numeral) \rightarrow np(NP), numeral(Numeral).$$
$$np(Prep@Noun) \rightarrow prep(Prep), noun(Noun).$$
$$vp(V) \rightarrow v(V).$$

with the lexical entries

$$noun(lambda(A, man(A)) \rightarrow [men].$$
$$v(lambda(B, cry(B))) \rightarrow [cry].$$
$$prep(lambda(C, C) \rightarrow [at]$$
$$noun(lambda(D, lambda(E, lambda(F, exists(N, X, E@X\&$$
$$F@X\&least(N, D))))))?[least]$$
$$numeral(lambda(G, G)@4) \rightarrow [four].$$

where $cry(B)$ expresses that B cry.

So if we have the following goal (in Prolog):

```
?-np(Sem,[At least four men cry],[]).
```

the `Sem` variable will be bounded to (we used bracket for a clear see):

```
Sem=(((lambda(C,C)@ lambda(D,lambda(E,lambda(F,exists(N,X,E@X&F@X&least(N,D))))))
   @ lambda(G,G)@4) @ lambda(A,man(A))) @ lambda(B,cry(B))
```

The β-conversion on the expression `Sem` will be:

```
1. Sem=((lambda(D,lambda(E,lambda(F,exists(N,X,E@X&F@X&least(N,D)))))
          @ lambda(G,G)@4) @ lambda(A,man(A))) @ lambda(B,cry(B))
2. Sem=((lambda(D,lambda(E,lambda(F,exists(N,X,E@X&F@X&least(N,D)))))
          @ 4) @ lambda(A,man(A))) @ lambda(B,cry(B))
3. Sem=(lambda(E,lambda(F,exists(N,X,E@X&F@X&least(N,4))))
           @ lambda(A,man(A))) @ lambda(B,cry(B))
4. Sem=lambda(F,exists(N,X, lambda(A,man(A))@X&F@X&least(N,4)))@
        lambda(B,cry(B))
5. Sem=lambda(F,exists(N,X, man(X)&F@X&least(N,4)))@ lambda(B,cry(B))
6. Sem=exists(N,X, man(X)& lambda(B,cry(B))@X&least(N,4))
7. Sem=exists(N,X, man(X)& cry(X))&least(N,4))
```

which represents the formula:

$$\exists_N X.(man(X) \wedge cry(X) \wedge least(N, 4))$$

With this method we can represent sentences like "At the most three men", "John eats maximum three apples", etc.

2.2. **Implementing Lambda Calculus.** Of course, now we want to reduce this complicated lambda expressions into readable first order formulas by carrying out β-conversion. The following code does this (see [1, 2]):

```
betaConvert(Var,Result):- var(Var),!, Result=Var.
betaConvert(Functor@Arg,Result):-
   compound(Functor),
   betaConvert(Functor,ConvertedFunctor),
   apply(ConvertedFunctor,Arg,BetaConverted),!,
   betaConvert(BetaConverted,Result).
betaConvert(Formula,Result):-
   compose(Formula,Functor,Formulas),
   betaConvertList(Formulas, ResultFormulas),
   compose(Result,Functor,ResultFormulas).
```

The first clause of the `betaConvert/2` simply records the fact that variable cannot be further reduced. The second clause does the most important things: it checks whether the functor is complex term and then reduces it to a lambda expression. If that succeeds, it applies the converted functor to `Arg` using `apply/3`.

The third and final clause breaks down formulas and predicates and reduces their arguments of subformulas. This is done by the help of:

```
betaConvertList([],[]).
betaConvertList([Formula|Others],[Result|ResultOthers]):-
   betaConvert(Formla,Result),
   betaConvertList(Others,ResultOthers).
```

In order to define the `apply/3` we first define the `substitute/4` predicate:

```
substitute(Term,Var,Exp,Result):-
   Exp=Var, !, Result=Term.

substitute(_Term,_Var,Exp,Result):-
   \+ compound(Exp) , !, Result=Term.
substitute(Term,Var,Exp,Result):-
   compose(Formula,Functor,[Exp,F]),
   member(Funcotr,[lambda, forall, exists]), !,
   (
   Exp=Var, !,
   Result=Formula
   ;
   substitute(Term,Var,F,R),
   compose(Result, Functor, [Expr,R])
).
substitute(Term, Var, Formula, Result):-
   Compose(Formula, Functor, ArgList),
```

```
   substituteList(Term, Var, ArgList, ResultList),
   compose(Result, Functor, ResultList).
substituteList(Term, Var, [], []).

substituteList(Term, Var, [Exp|Others], [Result| ResultOthers]):-
   substitute(Term,Var, Exp, Result),
   substituteList(Term, Var, Others, resultOthers).
```

Here is an example about the functionality of this predicate.

```
?-substitute(A,B,love(C,B), Result).

Result=love(C,A)
```

then `apply/3` will be

```
apply(lambda(X,Formula), Argument, Result):-
   substitute(Argument, X, Formula, Result).
```

To finish off, we define a driver predicate that calls the parser to analyze a sentence, reduces the resulting lambda expression into a first order formula, and directs the result to the standard output:

```
parse:-
   readLine(Sentence),
   s(LambdaExpression, Sentence,[]),
   normalform(LambdaExpression, NormalLambdaExpression),
   betaConvert(NormalLambdaExpression, Formula),
   printRepresentation(Formula).
```

Here the predicate `normalform(LambdaExpression, NormalLambdaExpression)` transform a `LambdaExpression` into a `normalform` labda expression, i.e. if we have the `LambdaExpression`: `exists(1,X,lambda(A, man(A))&room(X))@vincent` in normal form it will be `lambda(A, exists(1,X,man(A)&room(X))@vincent`.

References

[1] P. Blackburn, J. Bos, *Representation and Inference for Natural Language.* A first Course in Compuational Semantics, Volume I Working with first oreder logic. Computerlinguistik, Universitt des Saarlandes, September 1999.

[2] P. Blackburn, J. Bos, *Representation and Inference for Natural Language.* A first Course in Compuational Semantics, Volume II Working with discourse representation structures". Computerlinguistik, Universitat des Saarlandes, September 1999.

[3] D. Tatar ,*Bazele matematice ale calculatoarelor*, Curs lito. Universitatea "Babes-Bolyai" Cluj- Napoca, Facultatea de Matematica si Informatica, 1993.

[4] J. Allen, *Natural Language Understanding*, The Benjamin/ Cumings Publishing Company, Massachusetts, 1995

[5] D. Tatar, A. Onet, *Automated definite clause grammars compiling*, Presented at ROSYCS' 98 at the "Alexandru Ioan Cuza" University of Iasi, 1998.

"Babes-Bolyai" University, Faculty of Mathematics and Computer Science, RO-3400 Cluj-Napoca, Romania

E-mail address: {adrian,dtatar}@cs.ubbcluj.ro