ISSN 2007-9737

Algorithm for Processing Queries that Involve Boolean Columns for a Natural Language Interface to Databases

Rodolfo A. Pazos R., José A. Martínez F., J. Javier. Gonzalez B., Andrés A. Verástegui O.

Tecnológico Nacional de México,
Instituto Tecnológico de Ciudad Madero,
Mexico

r_pazos_r@yahoo.com.mx, jose.mtz@gmail.com,
{jjgonzalezbarbosa, andres.verastegui}@hotmail.com

Abstract. In the last decades, the use of natural language interfaces to databases (NLIDBs) has increased exponentially; unfortunately, the complexity of natural language has limited their effectiveness. The presence of Boolean columns in databases increases the difficulty for translating natural language queries to SQL. A Boolean column is a column that can only store two possible values: true/false, yes/no, 1/0. The problem for processing queries that involve Boolean columns, is that the search value for these columns (true/false, yes/no, 1/0) is not explicit in the queries. This problem makes NLIDBs generate erroneous translations as shown in experimental tests. A survey of the literature on NLIDBs has shown that this problem has not been identified, much less addressed. In this article, a new algorithm for processing queries that involve Boolean columns is presented. The algorithm uses syntactic and semantic information that facilitates detecting Boolean columns and their implicit values in a query. The experimental tests show that it is highly effective for translating this type of queries.

Keywords. Natural language interfaces to databases, natural language processing, databases, SQL.

1 Introduction

Currently, the fast growth in the use and size of databases (DBs) makes imperative to facilitate access to information. For accessing database information, different types of software tools have been developed. One type of such tools is DB query languages; for example SQL, which is widely used by computer professionals, but it is difficult to use by non-professionals. Another type is natural language interfaces to databases (NLIDBs), which originated in the decade of the 60s; unfortunately, the complexity of natural language (NL) and technological limitations hindered the advancement of NLIDBs. Currently, due to progress in computer and information technology, there is a powerful technological base for improving the performance of NLIDBs; however, the difficulty of natural language remains a major challenge. Treatment of search values in NLIDBs is a fundamental part in the translation of NL to a DB query language (such as SQL), since search values allow to specify the information to be looked for in the database. An important problem in the treatment of search values is determining the DB table and column where the value might be stored. The search values dealt for in the NLIDB described in this article can be classified in four different types:

– Easily detectable values: those that have particular characteristics that facilitate their detection. For example, proper nouns, integer and real numbers, codes, dates, and hours.

– Imprecise values: words that may refer to ranges, such as morning, evening, night, etc.

– Alias values: words that are different to the usual or formal term for referring to values. For example, Philly can be used for referring to Philadelphia.

– Boolean column values: those that can only have two values: true/false, yes/no, 1/0.

The problem posed by Boolean columns occurs when the query has a word or phrase that refers to a column of this type. In this case the query does not specify the search value (true/false, yes/no, 1/0). Therefore, the problem consists in detecting the Boolean column and the implicit search values, which are needed for generating the adequate search condition for the SQL statement. The following is an example of this type of queries to the ATIS (Air Travel Information Services) database [4]:

Which aircraft types are not wide body?

This query involves only table *aircraft* and a Boolean column, *wide_body*, which has two possible values: YES and NO. Neither of these values is present in the query; however, the NO value is implicit. The translation to SQL is the following:

SELECT *aircraft.aircraft_type, aircraft.wide_body*
FROM *aircraft*
WHERE *aircraft.wide_body* LIKE 'NO';

Additionally, more difficult cases are those where the query involves two or more Boolean columns, and the implicit search values might be positive (true, yes, 1) for some columns and negative (false, no, 0) for others. The previous version of the NLIDB [6] only processed correctly queries that included easily detectable, imprecise and alias values. This was so because of limitations of the syntactic parser of the previous version. This article presents a new syntactic-semantic method that allows to translate NL queries that involve Boolean columns.

The NLIDB described in this article was designed for answering queries in Spanish; however, the proposed approach for treating queries that involve Boolean columns is not particular for Spanish, so it can be applied to other languages: English, French, Italian and Portuguese. Since, the NLIDB is for Spanish many example queries will be written in Spanish and English.

2 Related Work

Most of the literature on NLIDBs does not describe how the interfaces deal with search values in the process of translating a NL query to SQL. Therefore, an exhaustive search of the literature from 2010 was carried out in order to find all the existing information on how NLIDBs are dealing with search values. Though all the interfaces have to deal with search values; unfortunately, only a few describe this aspect. Table 1 shows the NLIDBs, whose publications mention the treatment of search values. The table shows the ability of the interfaces for querying different databases (domain independence) and the method used for dealing with search values.

Table 1. Treatment of search values in NLIDBs

NLIDB	Domain independence	Treatment of values
ELF	✓	Dictionary
English	✓	Lexicon
Query		(automatically adds values)
Precise	✓	Lexicon (automatically adds values)
C-Phrase	✓	Domain dictionary

ELF (English Languaje Frontend) is a commercial NLIDB for querying different relational databases using natural language [3]. ELF Sofware claims it is the commercial interface with the best performance (recall). ELF provides domain independence; i.e., once ELF is installed on a computer, it can be used for any database, it is only necessary to configure it for each database.

ELF can be automatically configured by obtaining information from the DB schema; therefore, its initial configuration is fast and simple. During the translation process, ELF examines the terms (words/phrases) used for defining DB tables and columns, and it uses its dictionary for predicting synonyms used in queries. English Query (EQ) is a commercial NLIDB that was a part of Microsoft SQL Server 2000.

This application receives queries in English, determines their meaning, and generates and executes an SQL statement [1]. EQ has a dictionary with thousands of usual words in English. This dictionary provides the terminology necessary for answering most of the queries formulated in English. The dictionary can be expanded by creating entities (with synonyms) and relationships that provide specialized words for a specific application. Words related to tables, columns and values are automatically created in the dictionary.

Precise is a NLIDB considered as one of the most successful [8]. However, it can virtually only answer queries considered ”tractable”, where this term is defined as a query easy to understand, where words or phrases correspond to DB elements (tables, columns, relationships) or constraints. Precise stores in the lexicon values, column names and relationships from the database. The lexicon can be manually augmented by adding relevant synonyms, prepositions, etc. The lexicon search is performed token by token using the lemma of each token, this way it retrieves the matches that probably refer to database elements.

C-Phrase is a NLIDB that uses NL processing techniques [5]. To this end, it uses the following components: domain dictionary, formal semantic language, and GUI-based authoring tool. For validating the interpretation, it paraphrases queries to clarify to users what the system understood. C-Phrase uses synchronous context free grammars for analyzing NL queries.

Three methods are mentioned for dealing with search values by the NLIDBs in Table 1. Each of these techniques has advantages and disadvantages. The three basic methods for dealing with search values are the following:

– They are included in the interface dictionary.

– They can be recognized by a pattern.

– They are searched for and located in the database.

If search values are stored in the domain dictionary, they are extracted automatically from the database to be queried when the NLIDB is initially configured. In this case, for each value the dictionary stores the value, the table and column where it is stored. The main advantage of this technique is that it facilitates the identification of the table and column where the value is stored.

The main disadvantage is that this approach does not work well for dynamic databases (DBs whose data is frequently modified), because the dictionary is not automatically updated when DB data is modified; therefore, the dictionary has to be updated every now and then. Another disadvantage is that the technique cannot be used for large databases (tables with more than 100 thousand rows), because the dictionary size is proportional to the DB size. Finally, when a search value is stored in two or more columns, sometimes the NLIDB gets confused and chooses the wrong column for the translation to SQL.

Some search values (proper nouns, integer and real numbers, codes, dates, and hours) in NL queries can be recognized by a pattern. For example, if dates are written in the query as dd/mm/yyyy, they can easily be identified. The main advantage of this approach is that value detection is fast and effective, and there is no need to store this type of values in the NLIDB dictionary. A disadvantage is that it might be difficult to determine the DB table and column where the value might be stored.

Another method for detecting a value consists in searching the value in the database. Like the first technique, its main advantage is that it facilitates the identification of the table and column where the value is stored. Another advantage is that the size of the dictionary is not proportional to the DB size. Unlike the first approach, an advantage is that it can be used for dynamic databases. The main disadvantage is that this approach does not work well for large DBs, because, the time for locating a value is proportional to the DB size. Like the first technique, a disadvantage is that the interface might get confused when the search value is stored in two or more columns.The problem in processing queries that involve Boolean columns is that the search value is not specified in the query.

Therefore, from the analysis of the methods used by NLIDBs for dealing with search values, it can be concluded that they are not adequate for solving this problem, because they need that the value be specified in the query. This article proposes a new approach for dealing with this problem.

3 Description of the NLIDB

The translation process of the NLIDB is based on functionality layers for systematically solving the problems found in the translation from NL to SQL [6]. A main component of the interface is a semantically enriched dictionary, which is used for facilitating query interpretation.

The functionality layers of the interface are shown in Fig. 1. Therefore, the process of a query is carried out in a linear and systematic way. The lexical analysis layer tags each token of the NL query by finding the syntactic category (or categories) of the token in the lexicon, which ideally contains all the language words.

The syntactic parsing, in a previous version of the NLIDB, performed a shallow parsing of the tokens and assigned one syntactic category to those tokens that have several categories; for example, the word *lista* (*list*) that has three categories in Spanish. Additionally, the parser marked irrelevant words such as articles, verbs at the beginning of the query and some prepositions, so they were ignored in the semantic analysis.

The third layer, and the most important, performs the semantic analysis. This layer determines the meaning of the query by using the semantic information dictionary (SID), described in Subsection 3.1. More details of the translation process can be found in [6].

3.1 Semantic Information Dictionary

One of the most important elements of the NLIDB is the semantic information dictionary (SID), which contains most of the relevant information for the translation of queries. Initially, the information is introduced in the SID by an automatic configuration process of the interface, which extracts the information from the schema of the DB to be queried.

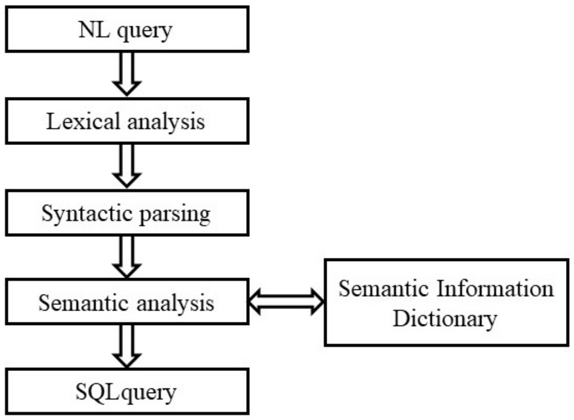


Fig. 1. Functionality layers of the translation process

Usually, this information allows the interface to obtain a recall (percentage of correctly answered queries) of just 17% [6].

In order to improve the recall, it is necessary that the database administrator fine-tune the configuration. The fine-tuning can be performed using a configuration editor, which allows to introduce in the SID linguistic terms (called descriptors) used in the domain of the database to be queried.

The SID contains descriptors (words and phrases) that usually occur in NL queries for referring to tables, columns and relationships between tables. The quality of the information stored in the SID influences directly on the recall obtained by the NLIDB. The SID is implemented as a relational database. Fig. 2 shows the schema of SID, configured for the ATIS database [4]. A more detailed description of the SID can be found in [6].

For processing queries that involve Boolean columns, a new column (*boolean_column*) was included in table *columns* of the SID. The new column is used for indicating if a column is Boolean or not. Therefore, during the semantic analysis, the SID can be searched for finding out whether or not a column is Boolean. For a DB column that is not Boolean, the value stored in *boolean_column* is null; otherwise, for a DB column that is Boolean, *boolean_column* stores the two possible values for the Boolean column (Fig. 3).

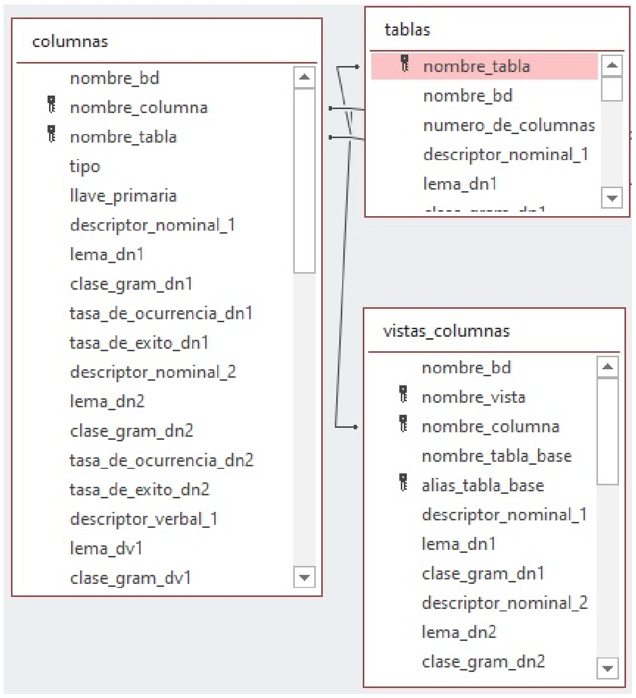
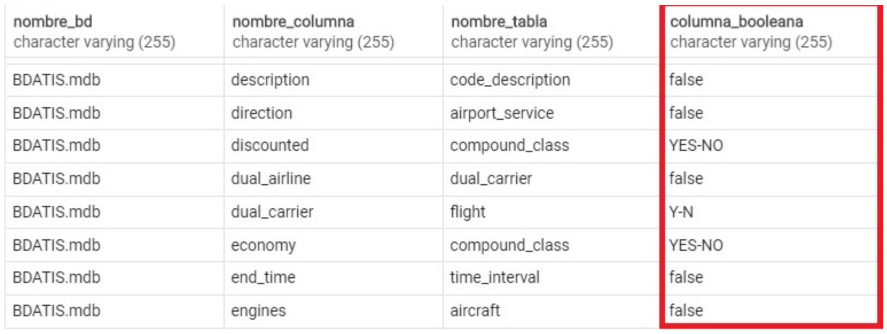


Fig. 2. Semantic information dictionary



nombre_bd character varying (255)	nombre_columna character varying (255)	nombre_tabla character varying (255)	columna_booleana character varying (255)
BDATIS.mdb	description	code_description	false
BDATIS.mdb	direction	airport_service	false
BDATIS.mdb	discounted	compound_class	YES-NO
BDATIS.mdb	dual_airline	dual_carrier	false
BDATIS.mdb	dual_carrier	flight	Y-N
BDATIS.mdb	economy	compound_class	YES-NO
BDATIS.mdb	end_time	time_interval	false
BDATIS.mdb	engines	aircraft	false

Fig. 3. Information of Boolean columns in the SID

This allows to obtain from the SID the two possible values for a Boolean column.

3.2 Semantic Analysis

The semantic analysis is the most important part of the translation process, its function is to understand what information from the database is requested by the query.

The semantic analysis consists of algorithms that detect what tokens refer to DB tables, columns and search values. The detection is performed using the SID, which stores the tokens (descriptors) used for referring to DB tables, columns and relationships between tables.

Fig. 4 shows the sublayers of the semantic analysis. The Treatment of Imprecise and Alias Values is a sublayer that detects imprecise values in the NL query and retrieves from the SID the

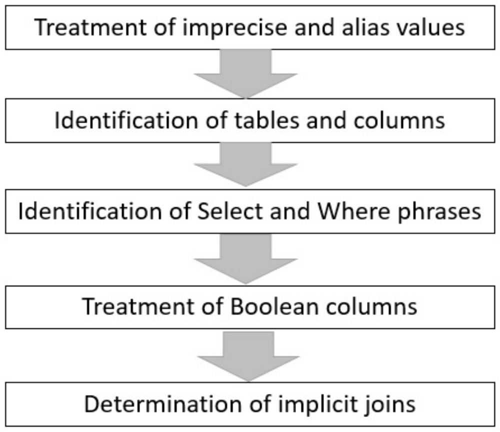

Treatment of imprecise and alias values
Identification of tables and columns
Identification of Select and Where phrases
Treatment of Boolean columns
Determination of implicit joins

Fig. 4. Sublayers of the semantic analysis

corresponding value range, and performs a similar process for alias values.

The Identification of Tables and Columns is the sublayer that detects tokens that might refer to DB tables and columns and retrieves their names from the SID. Additionally, this layer tags each token with the corresponding table and column name.

The Identification of the Select and Where Phrases is a sublayer that uses a heuristics for detecting the tokens that refer to DB columns that must constitute the Select and Where clauses of the SQL statement. In addition, this layer attaches Select or Where tags to the tokens identified. Finally, this sublayer relates columns to their respective search values specified in the NL query.

The Treatment of Boolean Columns is a sublayer that uses the information collected in the previous sublayers (Fig. 4) for detecting Boolean columns using the algorithms described in Section 5.

At this point in the semantic analysis, almost all the information has been collected for generating the SQL statement. The Determination of Implicit Joins sublayer is used when the tables (detected in the preceding sublayers) and their relationships with the other tables do not constitute a connected graph. In this case, it is necessary to generate, by using a heuristics, a connected graph by including tables that are not mentioned in the NL query, but are necessary for connecting the tables involved in the query. More details of the semantic analysis can be found in [6].

4 Syntactic Parser

In Spanish (like in English) a grammatical rule allows to reduce a nominal phrase followed by a prepositional phrase to a nominal phrase; for example, for the query *Which Delta flights depart to Washington at night*?, *Washington* and *at night* can be reduced. This reduction, though syntactically correct, is semantically incoherent, because *at night* is not a modifier of *Washington*, but a modifier of *depart*. This example shows that, for performing an effective parsing, it is necessary that the parser have rules that combine syntactic and semantic information. The new version of the NLIDB has a new parser, which includes syntactic and semantic information in the design of the syntactic rules.

Table 2 shows a small sample of the reduction rules. The second column of the table shows the rules and the first shows rule identifiers. In this table, NomP0 and NomP1 denote nominal phrases. Additionally, nou denotes noun, and adj_cal, adj_com, adj_dem, adj_ind, adj_num, adj_pos, and adj_sup denote respectively the following types of adjectives: descriptive, comparative, demonstrative, indefinite, quantitative, possessive, and superlative.

Table 2. Sample of reduction rules

ID	Reduction rules
SN01	<NomP0> ::= <nou> [<adj_cal> | <adj_dem> | <adj_sup> | <adj_pos>]
SN02	<NomP0> ::= [<adj_com> | <adj_cal>| <adj_sup> | <adj_pos> | <adj_dem> | <adj_num>] <nou>
SN03	<NomP1> ::= (<adj_ind> <art> | <art> | <adj_ind>)
SV03	<SVer1> ::= [<adv>] <ver> [<adv>]
Note:	the notation used for the rules is described in [9]

The new version of the parser generates reductions that are semantically coherent.

This makes the parser very efficient regarding processing time (0.3 seconds on average for parsing a query) and yields a large percentage of correctly parsed queries (above 95%). Reductions for phrases are crucial for dealing with Boolean columns, since it is necessary to determine which tokens are reduced to a nominal phrase whose words refer to a Boolean column; for example in the query *Which aircraft types are not wide body?*, the nominal phrase *wide body* refers to a Boolean column.

Additionally, there exist queries that involve two or more Boolean columns, and the implicit search values might be positive (true, yes, 1) for some columns and negative (false, no, 0) for others. In this case, it would be difficult to determine the implicit values for the columns without an adequate reduction of tokens into phrases.

The process for detecting search values was modified in the new version of the syntactic parser. In the previous version, each token was looked for in the lexicon, and if it was not found, the token was marked as a possible search value. This was time consuming because the lexicon is stored in hard disk. In the new version, value detection is performed at the beginning. All the tokens of the NL query are scanned in one pass, and regular expressions are used for detecting codes, dates, proper nouns, and numbers; tokens that are not identified as values are looked for in the lexicon.

5 Syntactic-Semantic Method for Boolean Columns

The main problem in queries that involve Boolean columns occurs because this type of query does not specify the search value, since it is implicit. The usual way in which people request information that involves a Boolean column does not include explicitly the search value (true/false, yes/no or 1/0).

Normally, the search value is implicit in the word/phrase that refers to the column. The following example shows a NL query to the ATIS database and its translation to SQL, where the Boolean column *dual_carrier* stores values Y or N for indicating if the airline is dual or not:

*¿Cuáles aerolíneas **no** son una **empresa dual?***
*Which Airlines are **not** a **dual carrier?***

SELECT *airline.airline_name*, *flight.dual_carrier*
FROM *airline*, *flight*
WHERE *airline.airline_code* = *flight.airline_code*
AND *flight.dual_carrier* LIKE 'N';

For this query, the presence of adverb *no* (*not*) indicates that the implicit search value for column *dual_carrier* is N. However, this value cannot be extracted from the query, because it is not explicit. The new syntactic parser provides the information necessary for processing queries that involve Boolean columns, since the reductions generated by the parser are used for detecting possible Boolean columns and their implied values.

The first phase of the process is to obtain the syntactic categories of the query tokens, which is carried out by the lexical analyzer. The syntactic categories are shown next, where adj_int denotes interrogative adjective, nou indicates noun, adv stands for adverb, ver denotes verb, art indicates article and adj_cal indicates descriptive adjective.

¿Cuáles aerolíneas no son una empresa dual?
[adj_int, nou, adv, ver, art, nou, adj_cal]

For the preceding query, the parser generates the sequence of reductions shown in Table 3. In the sequence, the symbols involved in the next reduction are shown underlined. In this example, the terminal symbols are the syntactic categories, and the non-terminal symbols are NomP0 and NomP1, which denote nominal phrases, and VerP1 for verbal phrase. Additionally, SN01, SN03 and SV03 denote identifiers for reduction rules (Table 2).

Specifically, the process for a Boolean column is performed as follows. All tokens of the query (words in step 0, Table 3) are scanned for finding in the SID if a word/phrase refers to a Boolean column; in this example *empresa dual*. Next, it is necessary to determine if the implicit value for the Boolean column is positive or negative. This process is carried out by algorithms 1 and 2, whose pseudocodes are presented next.

Table 3. Sequence of reductions

Step	Symbols (terminal and non-terminal)
0:	*Cuáles aerolíneas* ***no*** *son una empresa dual*
	[adj_int, nou, **adv**, ver, art, **nou**, adj_cal]
1:	SN01
	[adj_int, NomP0, **adv**, ver, art, **nou**, adj_cal]
2:	SN01
	[adj_int, NomP0, **adv**, ver, art, **NomP0**]
3:	SN03
	[adj_int, NomP0, **adv**, ver, **NomP1**]
4:	SV03
	[adj_int, NomP0, **VerP1**, **NomP1**]

Algorithm 1: Pseudocode for Boolean columns

```
//N_t is the number of tokens (words/values in step 0)
//Q_i is the i-th token
//N_k is the number of symbols in step k
//Reds is the set of symbols of all reduction steps
1:  procedure booleanColumns(Q, Reds)
2:  for i=0 to N_t-1 do
3:    if isBoolean(Q_i) then //Q_i is a token that refers
        to a Boolean column
4:      Q_BC ← Q_i //Save the token of Boolean column
5:      p_bool ← getSymbolPosition(Q_BC)
        //Save position of symbol corresponding to Q_BC
6       typeVal ← determinePosNeg(Reds,p_bool)
        //Determine neg. or pos. value for Q_BC
7       setColumnAsBoolean(Q_BC) //Change phrase
        type from Select to Where
8       addValueToken (Q_BC, typeVal) //Add a token
        with positive/negative value for Bool. column
9     end if
10  end for
```

The processing of queries that involve Boolean columns is a sublayer of the semantic analysis (Fig. 4) and uses information obtained during the syntactic parsing (sequence of reductions, Table 3) and the semantic analysis.

Specifically, it uses information generated by the Identification of Tables and Columns sublayer (Fig. 4), which detects tokens that might refer to DB tables and columns and retrieves their names from the SID.

Additionally, the Identification of the Select and Where Phrases sublayer attaches a Select tag to each of the tokens that refer to Boolean columns. Since this sublayer does not find a value in the query associated to a Boolean column, it assumes that the column must be in the column list of the Select clause of the SQL statement.

Algorithm 2: Pseudocode for determining positive or negative column values

```
//R is the number of reduction steps
//S_ki is the i-th symbol of step k
1:  procedure determinePosNeg(Reds, p_bool)
2:  for k = R-1 to 1 do
      //Scan reduction steps starting from the last
3:    for i = 0 to N_k-1 do //Scan reduction symbols
4:      if isReductionOfBoolCol(S_ki, p_bool) then
          //Symbol S_ki is reduction of Boolean column
5:        if S_ki is prepositional phrase then
            //Prepositional phrase includes Bool. col.
6:          Q_first ← getFirstToken(S_ki) //Get 1st. token
            of phrase
7:          if Q_first = 'sin' then return 'neg' //1st. token
              of phrase is sin (without), return negative
8:        end if
9:        if S_ki is nominal or adjectival phrase then
            //nom./adj. phrase includes Bool. col.
10:         if S_k,i-1 is verbal phrase then //nom./adj.
              phrase is preceded by verbal phrase
11:           Q_adv ← findAdverb(S_k,i-1) //Look for
              adverb in verbal phrase
12:           if Q_adv = 'no' then return 'neg'
                //Adverb in phrase is no (not), return
                negative
13:         end if
14:       end if
15:       if S_k,i is verbal phrase then //Verbal phrase
          includes Bool. col.
16:         Q_adv ← findAdverb(S_k,i-1) //Look for adverb
            in verbal phrase
17:         if Q_adv = 'no' then return 'neg' //Adverb in
              phrase is no (not), return negative
18:       end if
19:     end if
20:   end for
21: end for
22: return 'pos' //Positive value for Bool. col.
```

Algorithm 1 shows the pseudocode that describes the sublayer for the treatment of Boolean columns (Fig. 4). In the pseudocode, Q denotes the NL query introduced by the user, and Q_i is a token of query Q. For understanding algorithms 1 and 2, it is important to keep in mind that each token in step 0 (Table 3) has a corresponding terminal symbol; i.e., a syntactic category. For example, the terminal symbol for *empresa* (*carrier*) is nou, and the terminal symbol for *no* (*not*) is adv. Additionally, each non-terminal symbol (steps 1, 2, etc.) keeps a list of the positions of the terminal symbols that were reduced to the non-terminal symbol. For example, in step 2, the last non-terminal symbol (NomP0) keeps the positions of nou and adj_cal (positions 5 and 6 in step 0). Also, in step 4, non-terminal symbol VerP1 keeps the positions of adv and ver (positions 2 and 3 in step 0).

In Algorithm 1, each token is scanned (line 2). For each token of query Q, if Q_i is a token that refers to a Boolean column (identified in the SID), then the symbol position of the token is obtained (lines 4 and 5). At line 6, it is determined (Algorithm 2) if the implicit value for the Boolean column must be positive or negative. At line 7 the phrase type of the column is changed from Select to Where, which means that the column must be considered for the search condition of the SQL statement. At line 8, an extra token for the implicit value (true/false, yes/no or 1/0) is added to the query, according to the type (positive or negative) determined at line 6. For the query of the example, the implicit value is N, which is retrieved from the SID.

In Algorithm 2, each reduction step is scanned (line 2) from the last one down to step 1. Additionally, for a reduction step k, each symbol is scanned (lines 3 and 4) for finding out the non-terminal symbol S_{ki}, which is a reduction that includes the terminal symbol at position p_{bool} (corresponding to a Boolean column). Once the non-terminal symbol has been identified, it is convenient to determine first if the implicit value for the Boolean column must be negative. To this end, it is necessary to consider three cases for non-terminal symbol S_{ki}: prepositional phrase (line 5), nominal or adjectival phrase (line 9) and verbal phrase (line 15).

If S_{ki} is a prepositional phrase and the first token of the phrase is preposition *sin* (*without*), then this implies a negative value (lines 5 to 8). If S_{ki} is a nominal or adjectival phrase and it is preceded by a verbal phrase ($S_{k,i\text{-}1}$) that includes adverb *no* (*not*), then a negative value is implied (lines 9 to 14).

If S_{ki} is a verbal phrase and it includes adverb *no* (*not*), then this implies a negative value (lines 15 to 18). Finally, if none of the conditions for a negative value is satisfied, then the implied value for the Boolean column must be positive (line 22).

The query in Table 3 will be used for illustrating the processing of Boolean columns. In Algorithm 1, the tokens are scanned, and it is determined that the words *empresa dua*l (*dual carrier*) refer to a Boolean column, then the position (5 in step 0 of Table 3) of token *empresa* is saved in p_{bool} (lines 4 and 5). Next, Algorithm 2 is called, which determines that the implicit value for the Boolean column (whose token is at position p_{bool}) is negative (line 6), because of the presence of adverb *no* (*not*). Additionally, the phrase type of the Boolean column is changed from Select to Where, and a new token is added to the NL query for the implicit value, which is N for this example.

Given a Boolean column, Algorithm 2 finds out if the implicit value for the column must be positive or negative. To this end, the reduction steps (Table 3) are scanned from the last step down to step 1 (line 2), for finding a non-terminal symbol that is a reduction of the terminal symbol corresponding to the Boolean column. For this example, the algorithm finds (at line 4) that at step 4 the symbol S_{ki} = NomP1 is a reduction of token *empresa* (*carrier*) whose terminal symbol is at position p_{bool} = 5. Since S_{ki} is a nominal phrase, then the algorithm continues at line 9; next at line 10, it detects that the preceding symbol is a verbal phrase (VerP1), and at lines 11 and 12 it finds that the verbal phrase contains adverb *no* (*not*). Therefore, the implicit value for the Boolean column must be negative, which is returned to Algorithm 1.

6 Experimental Results

Experimental tests were conducted using the ATIS (Air Travel Information Services) database [4], which has information on flights, flight fares, aircrafts, airlines, airports and cities in USA. It was decided to use ATIS, because its database has several Boolean columns: 2 in table *aircraft*, 4 in table *compound_class*, 1 in table *flight*, and 1 in table *restriction*.

Table 4. Test results for the NLIDB

1	Cuáles vuelos son en aviones de cuerpo ancho SELECT *flight.flight_number*, *aircraft.wide_body* FROM *aircraft*, *flight* WHERE *aircraft.aircraft_code* = *flight.aircraft_code* AND *aircraft.wide_body* LIKE 'YES';	✓
2	Dame los códigos de vuelo para vuelos con empresa dual SELECT *flight.flight_code* FROM *flight* WHERE *flight.dual_carrier* LIKE 'Y';	✓
3	Muestra el número de motores para aeronaves que no tienen presurización SELECT *aircraft.engines*, *aircraft.aircraft_type*, *aircraft.pressurized* FROM *aircraft* WHERE *aircraft.pressurized* LIKE 'NO';	✓
4	Cuáles tipos de aeronave no son de cuerpo ancho SELECT *aircraft.aircraft_type*, *aircraft.wide_body* FROM *aircraft* WHERE *aircraft.wide_body* LIKE 'NO';	✓
5	Cuáles aerolíneas no son empresa dual SELECT *airline.airline_name*, *flight.dual_carrier* FROM *airline*, *flight* WHERE *airline.airline_code* = *flight.airline_code* AND *flight.dual_carrier* LIKE 'N';	✓
6	Dame la clase de servicio para vuelos con descuento SELECT *class_of_service.class_description*, *flight.flight_number*, *compound_class.discounted* FROM *class_of_service*, *flight*, *compound_class*, *flight_fare*, *fare* WHERE *flight.flight_code* = *flight_fare.flight_code* AND *flight_fare.fare_code* = *fare.fare_code* AND *fare.fare_class* = *compound_class.fare_class* AND *compound_class.base_class* = *class_of_service.class_code* AND *compound_class.discounted* LIKE 'YES';	✓
7	Dame las clases con descuento SELECT *compound_class.class_type*, *compound_class.discounted* FROM *compound_class* WHERE *compound_class.discounted* LIKE 'YES'	✓
8	Dame la clase de tarifa para vuelos de primera SELECT *fare.fare_class*, *flight.flight_number*, *compound_class.premium* FROM *fare*, *flight*, *compound_class*, *flight_fare* WHERE *flight.flight_code* = *flight_fare.flight_code* AND *flight_fare.fare* = *fare.fare_code AND fare.fare_class* = *compound_class.fare_class* AND *compound_class.fare_class* = *fare.fare_class* AND *compound_class.premium* LIKE 'YES';	✓
9	Cuáles aerolíneas son empresa dual SELECT *airline.airline_name*, *flight.dual_carrier* FROM *airline*, *flight* WHERE *airline.airline_code* = *flight.airline_code* AND *flight.dual_carrier* LIKE 'Y';	✓
10	Cuáles tipos de aeronave no son de cuerpo ancho y no tienen presurización SELECT *aircraft.aircraft_type*, *aircraft.wide_body*, *aircraft.pressurized* FROM *aircraft* WHERE *aircraft.pressurized* LIKE 'NO' AND *aircraft.wide_body* LIKE 'NO';	✓
11	Cuáles tipos de aeronave no son de cuerpo ancho y tienen presurización SELECT *aircraft.aircraft_type*, *aircraft.wide_body*, *aircraft.pressurized* FROM *aircraft* WHERE *aircraft.pressurized* LIKE 'YES' AND *aircraft.wide_body* LIKE 'NO';	✓
12	Cuáles tipos de aeronave son de cuerpo ancho y tienen presurización SELECT *aircraft.aircraft_type*, *aircraft.wide_body*, *aircraft.pressurized* FROM *aircraft* WHERE *aircraft.pressurized* LIKE 'YES' AND *aircraft.wide_body* LIKE 'YES';	✓
13	Dame la restricción para vuelos sin escala SELECT *restriction.restrict_code*, *flight.flight_number*, *restriction.stopovers* FROM *restriction*, *flight*, *fare*, *flight_fare* WHERE *restriction.restrict_code* = *fare.restrict_code* AND *fare.fare_code* = *flight_fare.fare_code* AND *flight_fare.flight_code* = *flight.flight_code* AND *restriction.stopovers* LIKE 'N';	✓
14	Dame la aplicación para vuelos con escalas SELECT *restriction.application*, *flight.flight_number*, *restriction.stopovers* FROM *restriction*, *flight*, *fare*, *flight_fare* WHERE *restriction.restrict_code* = *fare.restrict_code* AND *fare.fare_code* = *flight_fare.fare_code* AND *flight_fare.flight_code* = *flight.flight_code* AND *restriction.stopovers* LIKE 'Y';	✓
15	Vuelos tarifa de primera desde ATL a PIT SELECT *flight.fligh_number*, *fare.one_way_cost* FROM *flight*, *fare*, *compound_class* WHERE *fare.fare_class* = *compound_class.fare_class* AND *flight.to_airport* LIKE 'PIT' AND *compound_class.premium* LIKE 'YES';	✓
16	Cuáles vuelos tienen tarifa con descuento SELECT *flight.flight_number*, *fare.one_way_cost* FROM *flight*, *fare*, *compound_class* WHERE *fare.fare_class* = *compound_class.fare_class* AND *compound_class.discounted* LIKE 'YES';	✓
17	Cuáles vuelos tienen tarifa sin descuento SELECT *flight.flight_number*, *fare.one_way_cost* FROM *flight*, *fare*, *compound_class* WHERE *fare.fare_class* = *compound_class.fare_class* AND *compound_class.discounted* LIKE 'NO';	✓

Table 5. Test results for the NLIDB (continuation for Table 4)

18	Cuáles aerolíneas tienen tarifa de primera desde SFO a DFW SELECT *airline.airline_name*, *fare.rnd_trip_cost* FROM *airline*, *fare*, *compound_class*, *flight*, *restrict_carrier*, *restriction*, *flight_fare* WHERE *airline.airline_name* = *restrict_carrier.airline_code* AND *restrict_carrier.restrict_code* = *restriction.restrict_code* AND *restriction.restrict_code* = *fare.restrict_code* AND *fare.fare_class* = *compound_class.fare_class* AND *compound_class.fare_class* = *fare.fare_class* AND *fare.fare_code* = *flight_fare.fare_code* AND *flight_fare.flight_code* = *flight.flight_code* AND *flight.airline_code* = *airline.airline_code* AND *flight.to_airport* LIKE 'DFW' AND *flight.from_airport* LIKE 'SFO' AND *compound_class.premium* LIKE 'YES';	✓
19	Dame los códigos de vuelo para vuelos con empresa dual SELECT *flight.flight_code* FROM *flight* WHERE *flight.dual_carrier* LIKE 'Y'	✓
20	Dame todos los vuelos desde DFW a DEN de cuerpo ancho SELECT *flight.flight_number*, *flight.from_airport*, *flight.to_airport*, *aircraft.wide_body* FROM *flight*, *aircraft* WHERE *flight.aircraft_code* = *aircraft.aircraft_code* AND *aircraft.wide_body* LIKE 'YES' AND *flight.to_airport* LIKE 'DEN' AND *flight.from_airport* LIKE 'DFW';	✓

Table 6. Summary of resuts of the tests on ELF

Type of error	No. of quries
Incorrect translation	11
No translation (unable)	6
No translation	2
(words not understood or unfamiliar)	
No translation (not enough parameters)	1

Twenty queries than involve Boolean columns were taken from the ATIS query corpus. The query corpus contains queries that involve one or more columns whose implicit values are positive and negative, as shown in Table 4. The metric for evaluating performance is recall [7], which is defined as follows:

$$\text{recall} = \frac{\text{number of correct query answers}}{\text{number of queries}} \times 100.$$

The experimental results show that all the queries were correctly answered. Since the test involves only 20 queries, because the ATIS corpora has a small proportion of queries that involve Boolean columns and many queries are variations of other queries; therefore, we cannot claim that the recall is 100%, but we can conclude that the proposed approach is highly effective.

The experiments were executed on a laptop with an Intel Core i5 at 2.5 GHz with 4 GB of RAM and a Windows 10 operating system. The average processing time for the 20 queries was 0.548 sec., which is itemized as follows: 0.267 for the lexical analysis, 0.008 sec. for the syntactic parsing and 0.273 sec. for the semantic analysis.

The well-informed reader might notice that the query corpus for ATIS does not include queries involving airport codes, but city names; for example:

Dame los vuelos de Atlanta a San Francisco en la aerolínea DELTA AIRLINES primera clase

Give me the flights from Atlanta to San Francisco in the airline DELTA AIRLINES first class

The translation to SQL of this query generated by the NLIDB is the following:

SELECT*TblVistaTmp.flight_number*,
TblVistaTmp.CDO_city_name,
TblVistaTmp.CDD_city_name, *airline.airline_name*,
compound_class.premium,
compound_class.class_type
FROM *TblVistaTmp*, *compound_class*, *airline*,
fare, *restriction*, *restrict_carrier*
WHERE *compound_class.fare_class* =
fare.fare_class AND *fare.restrict_code* =
restriction.restrict_code AND
restriction.restrict_code =
restrict_carrier.restrict_code AND
restrict_carrier.airline_code = *airline.airline_name*
AND *airline.airline_code* =
TblVistaTmp.airline_code

Table 7. Detailed results of the tests on ELF

1	Which flights are in wide body airplanes The following words are not understood	**x**
2	Give me the flight codes for flights with dual carrier SELECT DISTINCT *flight.flight_code*, *dual_carrier.fconnection_name* FROM *airline*, *flight, dual_carrier*, *airline* INNER JOIN *flight* ON *airline.airline_code* = *flight.airline_code*, *airline* INNER JOIN *dual_carrier* ON *airline.airline_code* = *dual_carrier.main_airline* WHERE ((not (*dual_carrier.fconnection_name* Is Null or *dual_carrier.fconnection_name*="")));	**x**
3	Show me the number of engines for aircrafts with no pressurization Unfamiliar Word pressurization	**x**
4	What type of aircrafts are not wide body SELECT DISTINCT *aircraft.aircraft_type*, *aircraft.aircraft_code*, *aircraft.wide_body* FROM *aircraft* WHERE (((*aircraft.wide_body* Is Null or *aircraft.wide_body*="")));	**x**
5	Which airlines are not dual carrier SELECT DISTINCT *airline.airline_name*, *dual_carrier.fconnection_name* FROM *dual_carrier*, *airline*, *dual_carrier* RIGHT JOIN *airline* ON *dual_carrier.main_airline* = *airline.airline_code* WHERE ((*dual_carrier.fconnection_name* Is Null or *dual_carrier.fconnection_name*=""));	**x**
6	Give me the class type for flights with discount unable	**x**
7	Give me the classes with discount unable	**x**
8	Give me the fare class for premium flights SELECT DISTINCT *compound_class.fare_class*, *flight.flight_code*, *compound_class.PREMIUM* FROM *fare*, *compound_class*, *flight_fare*, *flight*, *fare* INNER JOIN *compound_class* ON *fare.fare_class* = *compound_class.fare_class*, *fare* INNER JOIN *flight_fare* ON *fare.fare_code* = *flight_fare.fare_code*, *flight_fare* INNER JOIN *flight* ON *flight_fare.flight_code* = *flight.flight_code* ORDER by *compound_class.PREMIUM*, *flight.flight_code*;	**x**
9	What airlines are dual carrier SELECT DISTINCT *airline.airline_name*, *dual_carrier.fconnection_name* FROM *dual_carrier*, *airline*, *dual_carrier* RIGHT JOIN *airline* ON *dual_carrier.main_airline* = *airline.airline_code* WHERE (not (*dual_carrier.fconnection_name* Is Null or *dual_carrier.fconnection_name*=""));	**x**
10	Which aircraft types are not wide body and are not pressurized SELECT DISTINCT *aircraft.aircraft_type*, *aircraft.pressurized*, *aircraft.wide_body*, *aircraft.aircraft_code* FROM *aircraft* WHERE (((*aircraft.wide_body* Is Null or *aircraft.wide_body*="") and (*aircraft.pressurized* Is Null or *aircraft.pressurized*="")));	**x**
11	Which aircraft type are not wide body and are pressurized SELECT DISTINCT *aircraft.aircraft_type*, *aircraft.pressurized*, *aircraft.wide_body*, *aircraft.aircraft_code* FROM *aircraft* WHERE (((*aircraft.wide_body* Is Null or *aircraft.wide_body*="") and not (*aircraft.pressurized* Is Null or *aircraft.pressurized*="")));	**x**
12	Which aircraft types are wide body and are pressurized SELECT DISTINCT *aircraft.aircraft_type*, *aircraft.pressurized*, *aircraft.wide_body*, *aircraft.aircraft_code* FROM *aircraft* WHERE ((not (*aircraft.wide_body* Is Null or *aircraft.wide_body*="") and not (*aircraft.pressurized* Is Null or *aircraft.pressurized*="")));	**x**
13	Give me the restrictions for flights with no stops SELECT DISTINCT *restriction.application*, *flight.flight_code*, *restrict_carrier.restrict_code* FROM *restrict_carrier*, *restriction*, *fare*, *compound_class*, *flight_fare*, *flight*, *airline*, *restrict_carrier* INNER JOIN *restriction* ON *restrict_carrier.restrict_code* = *restriction.restrict_code*, *restriction* INNER JOIN *fare* ON *restriction.restrict_code* = *fare.restrict_code*, *fare* INNER JOIN *compound_class* ON *fare.fare_class* = *compound_class.fare_class*, *fare* INNER JOIN *flight_fare* ON *fare.fare_code* = *flight_fare.fare_code*, *flight_fare* INNER JOIN *flight* ON *flight_fare.flight_code* = *flight.flight_code*, *flight* INNER JOIN *airline* ON *flight.airline_code* = *airline.airline_code* WHERE ((*compound_class.PREMIUM* = ”NO” and not (*flight.flight_code* Is Null or *flight.flight_code*=0)));	**x**
14	Give me the application for flights with stops unable	**x**
15	Premium class flights from ATL to PIT unable	**x**
16	Which flights have discounted fare not enough parameters	**x**

Table 8. Detailed results of the tests on ELF (continuation for table 7)

17	Which flights have no discounted fare SELECT DISTINCT *flight.flight_code*, *fare.fare_code* FROM *flight_fare*, *flight*, *fare*, *compound_class*, *flight_fare* INNER JOIN *flight* ON *flight_fare.flight_code* = *flight.flight_code*, *flight_fare* INNER JOIN *fare* ON *flight_fare.fare_code* = *fare.fare_code*, *fare* INNER JOIN *compound_class* ON *fare.fare_class* = *compound_class.fare_class* WHERE (*compound_class.DISCOUNTED* = ”NO”);	**x**
18	Which airlines have premium fare from SFO to DFW unable	**x**
19	Give me the flight codes for flights with dual carrier SELECT DISTINCT *flight.flight_code*, *dual_carrier.fconnection_name* FROM *airline*, *flight, dual_carrier*, *airline* INNER JOIN *flight* ON *airline.airline_code* = *flight.airline_code*, *airline* INNER JOIN *dual_carrier* ON *airline.airline_code* = *dual_carrier.main_airline* WHERE ((not (*dual_carrier.fconnection_name* Is Null or *dual_carrier.fconnection_name=””*)));	**x**
20	Give me all the wide body flights from DFW to DEN unable	**x**

AND*compound_class.premium* ILIKE ’YES’ AND *airline.airline_name* ILIKE ’DELTA AIRLINES’ AND *TblVistaTmp.CDD_city_nam* ILIKE ’San Francisco’ AND *TblVistaTmp.CDO_city_name* ILIKE ’Atlanta’;

Since the SQL statements for queries that involve city names is lengthy, for the test queries in Table 4, airport codes were used instead of city names.

In order to compare the performance of the NLIDB, the interface ELF was tested [3] using the same queries for the ATIS database. ELF was used because it is readily available for testing (unlike most NLIDBs) and is one of the few surviving commercial interfaces. The results are summarized in Table 6, and the detailed results are presented in the Tables 7 and 8.

According to the results obtained by ELF, its recall was 0%.

It is convenient to mention that it has been reported that ELF has a recall of 70-80% [2]. ELF fails in translating queries that involve Boolean columns, because there is no explicit search value in the queries, which shows the difficulty for dealing with this problem, not only for ELF, but for other NLIDBs that look for search values in the domain dictionary or the database.

Tables 7 and 8 show the results from the tests on ELF using the same queries as those in Tables 4 and 5. In this table, a cross indicates that the query was not translated or the translation was incorrect. The queries are in English, because ELF was designed for this language.

7 Conclusion and Future Work

The use of natural language interfaces to databases (NLIDBs) has not grown as fast as expected at the end of the 20th century, because of the difficulty present in natural language processing. Natural languages (NLs) have complex structures, whose understanding usually require using experience, intuition and clarification by human beings. Therefore, it is expected that the difficulty for understanding NL be much greater when using computational algorithms.

This article proposes an approach for translating NL queries that involve Boolean columns to an SQL statement. The new syntactic-semantic method constitutes a contribution in NLIDB research, showing that it is necessary to combine lexical, syntactic and semantic information, as human beings do, for obtaining high effectiveness.

The experimental results show that the proposed approach is highly effective (a recall close to 100%) for translating queries that involve Boolean columns. It is convenient to remark that a survey of the literature on NLIDBs has shown that this problem has not been identified, much less addressed (Section 2).

In order to keep increasing the recall of the NLIDB (which was 90% before the development of this new approach), another problem is being addressed: translating queries that involve search values that are language words; for example: *business class*, *breakfast*, *excursion fare*, and *coach* for the ATIS database. These values are

ISSN 2007-9737

difficult to identify by the NLIDB, because they are not easily detectable like proper nouns, codes or numbers. For solving this problem, the new syntactic parser is being used for detecting phrase patterns where this type of search values occur in queries, similar to the solution to the problem with Boolean columns.

Acknowledgements

The PhD student Andrés Verástegui acknowledges the scholarship (Grantee No. 429339) by the Consejo Nacional de Ciencia y Tecnología, Mexico.

References

1. **Bhootra, R. A. (2004).** *Natural Language Interfaces: Comparing English Language Front End and English Query*. Master's thesis, Virginia Commonwealth University.
2. **Conlon, S., Colon, J., & James, T. (2004).** The economics of natural language interfaces: natural language processing technology as a scarce resource. *Decision Support Systems*, Vol. 38, No. 1, pp. 141–159.
3. **ELF (2009).** *ELF Software Analyze*. http://www.elfsoft.com/help/accelf/Analyze.htm.
4. **Linguistic Data Consortium (1990).** *The 2884 ATIS0 Speaker-dependent Training Prompts*. http://www.ldc.upenn.edu/Catalog/readme_files /atis/sdtd/trn_prmp.html.
5. **Minock, M. (2010).** C-Phrase: A system for building robust natural language interfaces to databases. *Data & Knowledge Engineering*, Vol. 5, No. 3, pp. 290–302.
6. **Pazos, R. A., Aguirre, M. A., González, J. J., Martínez, J. A., Pérez, J., & Verástegui, A. A. (2016).** Comparative study on the customization of natural language interfaces to databases. *Springer-Plus*, Vol. 5, No. 1, pp. 553.
7. **Pazos, R. A., González, J. J., Aguirre, M. A., Martínez, J. A., & Fraire, H. J. (2013).** Natural language interfaces to databases: An analysis of the state of the art. In *Recent Advances on Hybrid Intelligent Systems, Studies in Computational Intelligence*, volume 451. Springer Berlin Heidelberg, pp. 463–480.
8. **Popescu, A.-M., Armanasu, A., Etzioni, O., Ko, D., & Yates, A. (2013).** Modern natural language interfaces to databases: composing statistical parsing with semantic tractability. In *Recent Advances on Hybrid Intelligent Systems, Studies in Computational Intelligence*, volume 451. Springer Berlin Heidelberg, pp. 463–480.
9. **World Wide Web Consortium (1982).** *BNF Notation for Syntax*. https://www.w3.org/ Notation.html.

Article received on 25/11/2019; accepted on 06/01/2020.
Corresponding author is Rodolfo A. Pazos R.